UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Vote électronique : définitions et techniques d'analyse

Electronic Voting: Definitions and Analysis Techniques

THÈSE

présentée et soutenue publiquement le 8 Novembre 2019

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Joseph Lallemand

Composition du jury

Gilles Barthe	Directeur Scientifique, MPI Bochum
Karthikeyan Bhargavan	Directeur de Recherche, Inria
Sandrine Blazy	Professeur, Université de Rennes
Ralf Kuesters	Professeur, Université de Stuttgart
Christine Paulin	Professeur, Université Paris-Sud – *Présidente du jury*
Emmanuel Thomé	Directeur de Recherche, Inria
Véronique Cortier	Directrice de Recherche, CNRS – *Directrice de thèse*

Rapporteurs

Gilles Barthe	Directeur Scientifique, MPI Bochum
David Pointcheval	Directeur de Recherche, CNRS

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Abstract

In this thesis we study several aspects of the security of remote electronic voting protocols. Such protocols describe how to securely organise elections over the Internet. They notably aim to guarantee vote privacy – *i.e.*, votes must remain secret – and verifiability – it must be possible to check that votes are correctly counted. Our contributions are on two aspects.

First, we propose a new approach to automatically prove equivalence properties in the symbolic model. Many privacy properties can be expressed as equivalence properties, such as in particular vote privacy, but also anonymity or unlinkability. Our approach relies on typing: we design a type system that can typecheck two protocols to prove their equivalence. We show that our type system soundly implies trace equivalence, both for bounded and unbounded numbers of sessions. We compare a prototype implementation of our typechecker with other existing tools for symbolic equivalence, on a variety of protocols from the literature. This case study shows that our procedure is much more efficient than most other tools – at the price of losing precision (our tool may fail to prove some equivalences).

Our second contribution is a study of the definitions of privacy and verifiability – more precisely, individual verifiability, a property that requires each voter to be able to check that their own vote is counted. We prove that, both in symbolic and computational models, privacy implies individual verifiability, contrary to intuition and related previous results that seem to indicate that these two properties are opposed. Our study also highlights a limitation of existing game-based definitions of privacy: they assume the ballot box is trusted, which makes for significantly weaker guarantees than what protocols aim for. Hence we propose a new game-based definition for vote privacy against a dishonest ballot box. We relate our definition to a simulation-based notion of privacy, to show that it provides meaningful guarantees, and conduct a case study on several voting schemes.

Keywords: Security, cryptographic protocols, electronic voting, equivalence, type systems, privacy

Résumé

Cette thèse porte sur l'étude de différents aspects de la sécurité des protocoles de vote électronique à distance. Ces protocoles décrivent comment organiser des élections par Internet de manière sécurisée. Ils ont notamment pour but d'apporter des garanties de secret du vote, et de vérifiabilité – *i.e.*, il doit être possible de s'assurer que les votes sont correctement comptabilisés. Nos contributions portent sur deux aspects principaux.

Premièrement, nous proposons une nouvelle technique d'analyse automatique de propriétés d'équivalence, dans le modèle symbolique. De nombreuses propriétés en lien avec la vie privée s'expriment comme des propriétés d'équivalence, telles que le secret du vote en particulier, mais aussi l'anonymat ou la non-traçabilité. Notre approche repose sur le typage: nous mettons au point un système de typage qui permet d'analyser deux protocoles pour prouver leur équivalence. Nous montrons que notre système de typage est correct, c'est-à-dire qu'il implique effectivement l'équivalence de traces, à la fois pour des nombres bornés et non bornés de sessions. Nous comparons l'implémentation d'un prototype de notre système avec les autres outils existants pour l'équivalence symbolique, sur divers protocoles de la littérature. Cette étude de cas montre que notre procédure est bien plus efficace que la plupart des autres outils – au prix d'une perte de précision (notre outil peut parfois échouer à prouver certaines équivalences).

Notre seconde contribution est une étude des définitions du secret du vote et de la vérifiabilité – ou, plus précisément, la vérifiabilité individuelle, une propriété qui requiert que chaque votant soit en mesure de vérifier que son propre vote a bien été pris en compte. Nous prouvons, aussi bien dans les modèles symbolique que calculatoire, que le secret du vote implique la vérifiabilité individuelle, alors même que l'intuition et des résultats voisins déjà établis semblaient indiquer que ces deux propriétés s'opposent. Notre étude met également en évidence une limitation des définitions existantes du secret du vote par jeux cryptographiques : elles supposent une urne honnête, et par conséquent expriment des garanties significativement plus faibles que celles que les protocoles visent à assurer. Nous proposons donc une nouvelle définition (par jeu) du secret du vote, contre une urne malhonnête. Nous relions notre définition à une notion de secret du vote par simulation, pour montrer qu'elle apporte des garanties fortes. Enfin, nous menons une étude de cas sur plusieurs systèmes de vote existants.

Mots-clés: Sécurité, protocoles cryptographiques, vote électronique, équivalence, typage, secret du vote

Contents

Introduction

Introduction

1 Context

Electronic voting has been gaining more and more importance in recent years. Information technology is more and more prominent in all domains of society, and it is then natural to envisage applying it to facilitate organising elections. Electronic voting – the use of computers to organise elections – manifests itself in two main forms. First, voting machines that are installed at polling stations, and that voters use to cast their votes. Second, remote voting systems, where voters cast their votes from the Internet. In this thesis, we will be more concerned with that second category.

The advantages of electronic voting are clear: it makes elections much more practical to organise than paper voting. In particular, votes are much easier to gather and count: ballots can simply be sent over the Internet to a server controlled by the authorities, and then tallied by computers. This could help reduce the infrastructure required to hold elections or referendums. Electronic voting systems, both on-site and remote, have already been deployed in real elections, both on small and large scales, *e.g.* Internet voting to elect the members of parliament in Estonia and France (in the latter case only for citizens living abroad) or for referendums in Switzerland, voting machines for national elections in the United States or in Brazil, etc.

But, as with any system that manipulates sensitive data, comes the problem of ensuring this data is securely handled. How can I be sure that the result announced by the election organisers is the right one? That my vote was properly counted, and not somehow removed from the ballot box? Can I be certain that my vote remains secret? That no one can learn in any way which candidate I voted for? Such questions have been discussed over centuries in the context of paper voting, and are typically considered crucial from the point of view of democracy. Indeed, what legitimacy would the winner have when there is no way to ensure they are the rightful winner? And how could I freely express my opinion by voting without fearing reprisal, if there was a risk that my vote could be revealed later on?

In paper-based election systems, it is possible, to a reasonable extent, to get these guarantees. At the polling station, I can secretly choose which ballot to put in an envelope and then in the ballot box. I can then watch the ballot box to check that my ballot is not opened before tallying. This allows me to ensure my vote remains secret. I can also oversee the tallying at the local polling station, to check that votes (including mine) are correctly counted.

However, it is often much less clear how to get such security with remote electronic voting systems. Still, it is in that case even more crucial to ensure the system is secure: since the election is held over the Internet, an attack on the system could be carried out from anywhere on a large scale, whereas in paper elections, manipulation of the votes would need to be performed in many polling stations to influence the result, which would be much harder to do undetected.

Protocols To provide such guarantees, remote voting systems take the form of cryptographic protocols. These make use of cryptography (*e.g.* encryption, signatures, ...) to protect the votes, and describe precisely what cryptographic operations should be performed by each party (*i.e.* the voters, authorities...). But it is now a well-known issue that cryptographic protocols are hard to get right – to make really secure. A major example is the TLS (Transport Layer Security) protocol. TLS aims to secure communications on the web. It notably provides authentication guarantees, so that users can be sure they are communicating with the server that they wanted to reach. It also aims to guarantee the confidentiality and integrity of communications: any data exchanged between a server and a client should remain secret from any other third party, and it should not be possible for such a third party to tamper with it. TLS was first proposed in 1999 [73], and then updated and corrected many times, the latest version being published in 2018 [92]: over time many vulnerabilities and attacks against each successive version have been found (*e.g.* Logjam [12], POODLE [84], Triple Handshake [30], ...), meaning that the protocol was never really secure.

Formal analysis Formal methods have proved to be a powerful tool to study the security of cryptographic protocols. Broadly speaking, they consist in designing mathematical models of the protocols, and of an adversary trying to attack them, as well as formal definitions for the security properties that should be guaranteed. The goal is then to prove formally that no adversary, in the chosen model, can make it so that the properties are violated. Over the last decades, they have been used to prove security or find attacks for many real-life protocols, such as the BAC (Basic Access Control) protocol used by RFID chips in biometric passports [14], the AKA (Authenticated Key Agreement) protocol used by mobile phones to connect to 3G/4G/5G networks [16, 23, 85], or the TLS protocol mentioned earlier [72].

Let us look at an example of a voting protocol, to illustrate the kind of attacks they can be subject to, and how formal methods can help in that context.

2 Example: Helios

Description Helios is an electronic voting protocol that was first proposed in 2009 [10]. Although it has not been deployed in large scale high-stake elections, it has been used in various smaller scale situations, such as the election of the president of the University of Louvain-la-Neuve [11].

Among other properties, Helios aims to protect the privacy of voters, by keeping their votes secret. To that end, Helios makes use of asymmetric cryptography. Technical details regarding the cryptographic construction used are irrelevant here, we will only describe informally its behaviour. Basically, an election authority, composed of several trustees, jointly generate a pair of a public and a private key for the election. The public key is published, while the private key is split into several shares that are distributed among the trustees. These shares are generated in such a way that all of the trustees (or a certain threshold of them) need to agree in order to decrypt a message. This decreases the risk that the talliers misuse the key: only a fraction of them needs to be trusted. The public key will then be used by the voters to encrypt their votes, and the trustees must agree to jointly decrypt the ballots, and compute the result. For better clarity, we will from now on in our explanations refer to the group of trustees as a if it were a single authority, which we will call election authority or tallying authority, that is the only owner of the secret key sk, and uses it to decrypt the ballots.

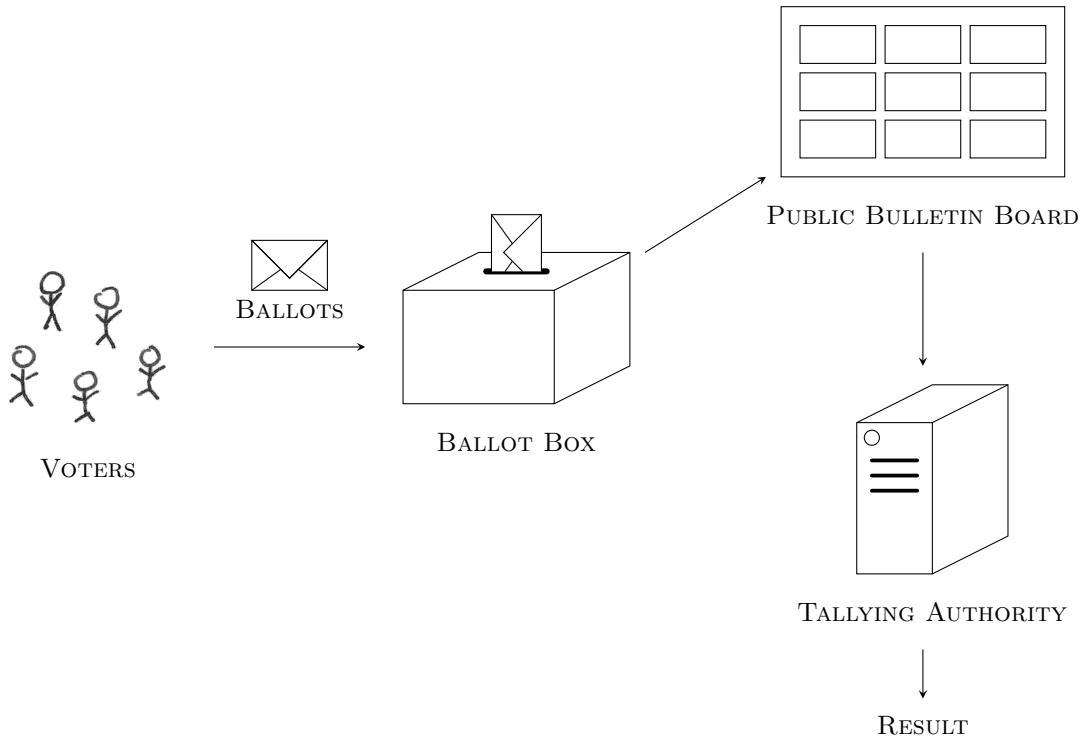


Figure 1 – The Helios voting protocol

The basic structure of an election in Helios is displayed in Figure 1, and (informally) described below.

- The election authority generates a public and private key pair $(\mathsf{pk}, \mathsf{sk})$, and publishes the public key pk. This is an abstraction of the real protocol used to generate and distribute the private key shares between the trustees.
- The voters construct ballots containing an encryption of their vote with pk.
- A voting server, that plays the role of a ballot box, collects the ballots from all voters. Concretely, voters log on to the server using some pre-established password, and send their ballot.
- Each time an encrypted ballot is received during the voting phase of the election, the voting server displays it on a public bulletin board, for everyone to see. In particular, voters can check, if they want to, that their ballot is there.
- Finally, the tallying authority (composed of all trustees) is in charge of computing the result of the election. It retrieves all ballots from the bulletin board, decrypts them (using the secret key sk), and then computes and publishes the result.

Attack At first glance, the secrecy of votes seems to be respected, provided the tallying authority is trusted. Indeed, any vote that is sent over the network is encrypted at all times, and only the talliers have (shares of) the key necessary to decrypt it. Hence, unless the cryptographic scheme used to encrypt votes is broken, no attacker seems to be able to somehow access the votes in clear.

However, there is actually an attack against this protocol, that was first described in [65]. Consider a very simple scenario where only three voters participate in the election: Alice, who wishes to express her vote for candidate α, Bob, who wants to vote for another candidate β, and Charlie, who is actually an attacker, and is trying to learn who Alice votes for. Following the protocol, Alice and Bob will respectively submit ballots b_A, b_B containing encryptions of their votes α, β to the ballot box.

If the election stopped at that point, the ballot box would contain only b_A and b_B. The result of the election would then be "one vote for α, one for β". When seeing this result, Charlie would not know whether it was Alice or Bob who voted for α: to learn Alice's vote, he has to take some additional actions.

Charlie could first eavesdrop on Alice when she sends her ballot b_A, and learn what this ballot is. He is then of course unable to decrypt it, since he does not have access to the secret key of the election. However, Charlie can instead submit to the ballot box a second copy of b_A, under his own name, *i.e.* pretending it is his own ballot, even though he actually does not know what this ballot contains. At that point, the list of all ballots in the ballot box is $[b_A, b_B, b_A]$. Opening the ballots, the tallying authority would then compute the result "two votes for α, one for β". This time, even if Charlie does not know what vote he submitted, he knows that that vote is the same as Alice's. Once the result is published, he would see that the only possibility is that that vote was actually α, since only that candidate got two votes. He would thus deduce that Alice wanted to vote for α: her privacy is breached.

This attack may seem simplistic, and difficult to apply in a real setting with many voters. However, the authors in [65] studied that question, and concluded that this attack (or variants of it) would actually be feasible in real elections, *e.g.* the French legislative elections, if that version of Helios was used for them.

Fix Fortunately, this ballot copying attack can be quite easily prevented. It relies on the ability for Charlie to submit an exact copy of Alice's ballot to the ballot box. The ballot box could easily detect that the same ballot has already been submitted by Alice earlier, and reject it. This correction to the protocol was proposed by the authors in [65]: they call the operation of rejecting these duplicate ballots *weeding*. It at least prevents the attack: now when Charlie tries to submit b_A in his own name to the ballot box, his attack is detected, and the ballot rejected. But is it actually sufficient to make Helios really secure? How can we know that there are no other ways of attacking the voters' privacy? This attack illustrates that even very simple flaws in a voting protocol are not at all obvious. This is where formal methods can help: by defining precisely what properties should be satisfied, we can formally verify that the protocols are secure.

In fact, as it turns out, performing weeding fails to prevent a similar attack discovered by Roenne, in a context where revote is allowed, *i.e.* when each voter can submit several ballots, of which only the last one will be counted. This second attack is as follows. The attacker Charlie could decide to not only eavesdrop, but also block Alice's ballot b_A entirely – cutting her connection to the voting server. She will try to vote again, with a new ballot b'_A for the same candidate. As before, Charlie will submit b_A under his own name. Even with weeding, the server will accept this ballot. Indeed, since b_A was blocked when Alice tried to submit it, the server has never seen that ballot. Hence, the result will, as in the previous attack, contain two votes for Alice's choice, and Charlie will thus learn her vote.

3 Properties

Different properties have been proposed to characterise the security guarantees an electronic voting protocol should provide. We consider here the following properties, that are arguably the most important ones.

Verifiability Intuitively, verifiability is the idea that people should be able to convince themselves that the result published for the election is the correct one, and has not been tampered with. It is typically formalised as the conjunction of the following three subproperties.

- Individual verifiability: I can check that *my* vote was correctly placed in the ballot box.
- Universal verifiability: Anyone can check that the result of the election correctly corresponds to the tally of the ballot box.
- Eligibility verifiability: Anyone can check that all ballots recorded in the ballot box were cast by legitimate voters.

Altogether, these three properties basically ensure that it is possible to check that the result published by the tallying authority correctly counts all votes from registered voters, and only those.

Privacy This property can informally be described as the guarantee that votes remain secret. However, this cannot simply be formalised by requiring that, when considering any arbitrary voter Alice, no attacker can learn the value of her vote. While such a condition can make sense when dealing with actual secrets (*e.g.* a key in a key exchange protocol), here the value v of the vote is simply one of the choices publicly available to voters in the election. The value of the vote itself is hence not secret. Rather, what is secret is the association between the voter and the vote: the fact that it was *Alice*, and not some other voter, who voted for v.

This is typically formalised as an indistinguishability property between two scenarios: one where Alice indeed votes for v, and one where some other voter Bob votes for v instead. More precisely, the condition is that for any arbitrary voters Alice, Bob and votes 0, 1, no attacker should be able to determine, by interacting with the voting protocol, whether Alice votes for 0 and Bob for 1, or Alice votes for 1 and Bob for 0. This indistinguishability property means that the attacker cannot know who voted for which candidate.

Receipt-freeness/coercion resistance Receipt-freeness and coercion resistance are other properties frequently required of voting protocols. They can be seen as stronger notions of privacy, as they express that my vote should remain secret *even if I am willing to reveal it.* That is, even if I am trying to reveal to some third party who I voted for, I should not be able to convince them I am telling the truth. More precisely, receipt-freeness states that, after I have voted, even if an attacker forces me to provide all the information I have regarding the election (*e.g.* depending on the protocol, my vote, my ballot, my credentials, any confirmation code I may have received...), he should not be able to determine whether I truthfully provide my real vote, or a fake one: the election process should not produce any receipt for my vote that I could show to someone else. Coercion-resistance further requires that the attacker still cannot learn my vote even if I am asked for all my information before I vote.

These strong properties provide protection against coercion, and vote selling: since there is no way for a third party to be sure I followed their instructions, they have no way of forcing me to vote for a specific candidate (either by coercing me, or even if I willingly sell my vote).

4 Relations between properties

An interesting point to note regarding the properties of voting systems we described earlier is that vote privacy intuitively seems to be at odds with verifiability. Indeed, vote privacy basically requires that as little information as possible leaks regarding the votes. Of course, the result of the election itself intrinsically gives some indication of how voters voted in any reasonable voting scheme; but privacy would require that this inevitable leakage is the only information an attacker can obtain on the individual votes. On the other hand, if a protocol is to be verifiable, it must include some mechanisms that allow voters to check that their vote was taken into account. To perform this verification, the voters typically need *some* information to be published by the authorities in addition to the result of the election. This additional information risks leaking information about the votes, which would violate vote privacy. The intuition hence seems to indicate that the more a voting system is designed with verifiability in mind, the harder it will be for this system to also guarantee vote privacy.

In fact, this intuition has been formally proved by Chevallier-Mames, Fouque, Pointcheval, Stern and Traoré [49]. They established that verifiability and unconditional privacy are incompatible. They consider a very strong notion of privacy, unconditional privacy, which requires that the votes remain perfectly secret even from an adversary that has unbounded computational power. However, no such result has been established in the case – usual for cryptographic protocols – of a polynomial adversary, such as is considered in all existing game-based definitions for vote privacy [26]. Still, following this intuition, official regulations and recommandations about electronic voting tend to favour one of the two properties over the other – for instance French recommandations only require vote privacy [2], though newer recommandations also ask for transparency [4], while Swiss law first requires privacy, and a level of verifiability that depends on the number of voters who vote online [3].

5 Formal analysis

To analyse a protocol *w.r.t.* the properties above, we need to design a formal model of both the protocol and the properties. We also need to specify against what kind of attacker they should hold. Several such models for security protocols have been proposed. They can mostly be sorted into two broad classes.

Symbolic models The attack on Helios described above stems from a logical flaw in the specification of the protocol, rather than an attack on the cryptographic primitives used: it can be performed without breaking the cryptography. Symbolic models focus on such attacks: they consider an abstract and idealised model of cryptography. While the security of the real cryptographic primitives usually relies on computational hardness assumptions (*e.g.* factorisation or discrete logarithm), in symbolic models, the primitives are assumed to be perfect and unbreakable. For instance, an encryption primitive will be assumed not to leak *any* information regarding the plaintexts to anyone who does not know the correct decryption key. Primitives are typically represented as abstract function symbols, and the messages exchanged by the different parties are modelled as abstract terms constructed using these symbols. The properties of the cryptographic primitives, *i.e.* the computations that the protocol parties and the adversary can perform on these terms, are characterised as an equational theory, that is, a set of equations that define the behaviour of the primitives.

For instance, asymmetric encryption could be modelled using a symbol `aenc`, with arity 2,

and a symbol $\mathtt{pk}$, with arity 1: $\mathtt{pk}(k)$ represents the public key associated with private key k, and $\mathtt{aenc}(M, \mathtt{pk}(k))$ represents the asymmetric encryption of message M with that public key. An associated symbol $\mathtt{adec}$ can be used to represent the corresponding decryption operation: $\mathtt{adec}(M, k)$ represents the operation of decrypting message M with the private key k.

The behaviour of the primitive would then be modelled as the following equation, stating that decrypting a message with the correct key successfully returns the original plaintext:

$$\mathtt{adec}(\mathtt{aenc}(M, \mathtt{pk}(k)), k) = M$$

This equational theory specifies that the only computation that can be performed on a ciphertext $\mathtt{aenc}(M, \mathtt{pk}(k))$ is to decrypt it using the key k. This models a perfect encryption primitive, that cannot be broken or leak information on the plaintext in any way to someone who does not have the decryption key.

Symbolic models typically consider an attacker that has complete control over the network, and can therefore intercept or block any messages sent by a protocol participant, and, within the limits allowed by the equational theory considered, use them to perform any computation, *i.e.* apply any function symbol to them to construct new terms. Finally the attacker has the ability to modify messages before their reach their destination, by replacing them with any term he can compute. This attacker model is usually called the *Dolev-Yao attacker*, as it was first considered by Dolev and Yao in 1983 [74].

Different symbolic models exist to describe the behaviour of protocols against such an attacker. One of the most prominent approaches, and the one we choose in this thesis when considering a symbolic setting, is to model protocols as processes in a process algebra such as the pi-calculus, which is a common tool to model distributed programs. This idea first took the form of the spi-calculus that was proposed by Abadi and Gordon in 1997 [8]. Standard pi-calculus provides constructions to model processes running in parallel, and exchanging simple messages by inputting or outputting them on communication channels. The spi-calculus extends it with cryptographic primitives, modelled as terms as explained earlier. It was then generalised into the applied pi-calculus by Abadi and Fournet in [6], which, compared to the spi-calculus, further separates the equational theory from the process algebra. This makes it easier to state security properties independently of the equational theory considered. Among other approaches are most notably those based on multiset rewriting [42], where protocols are represented as rewrite rules, that describe the evolution of the state of participants and the knowledge of the attacker during the execution of the protocol.

Computational models On the other hand, computational models, first explored in the 1980's (by *e.g.* Goldwasser, Micali [79], Blum [36], Yao [97]), consider an attacker model that is much less abstract, more precise, and arguably closer to the real world. Protocols are represented as a collection of algorithms (*i.e.* Turing machines) that describe the behaviour of each participant, and manipulate bitstrings, rather than abstract terms. Contrary to the symbolic case, the attacker is not explicitly restricted to performing certain computations specified by the primitives. Rather, the attacker is a probabilistic Turing machine, that can perform any arbitrary computations on the messages it observes, with the restriction that it must run in polynomial time *w.r.t.* a *security parameter* λ. That parameter typically represents the size of the keys used. Several ways of writing security properties in such models exist. *Simulation-based properties* – notably in the Universal Composability framework [41] – require that, in the eyes of the outside world, even in the presence of an adversary, the protocol being studied is indistinguishable from an ideal system whose security is manifest. *Game-based properties*, on the other hand, are expressed as

cryptographic games, where an adversary interacts with the protocol through oracles that run the algorithms modelling it, and is asked to try to break its security in a specific way. For instance, an indistinguishability property between two scenarios would be modelled by a game $\mathsf{Exp}^{\beta}(\lambda)$, where the adversary interacts with one of the two possible scenarios, depending on a secret bit β. The adversary is then asked to guess this bit β, *i.e.* to determine which scenario is running, and his guess constitutes the result of the game. The *advantage* of the adversary is then the difference between the probabilities it guesses 1 when $\beta = 0$ and when $\beta = 1$:

$$\mathsf{Adv}_{\mathcal{A}}(\lambda) = |\mathbb{P}(\mathsf{Exp}^{0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{1}(\lambda) = 1)|$$

Security is then defined by requiring that asymptotically (*w.r.t.* the security parameter λ), no adversary can do significantly better than random guessing. That is, that no adversary has a non-negligible advantage $\mathsf{Adv}_{\mathcal{A}}(\lambda)$ in that game.

As explained earlier, in the case of vote privacy, the two scenarios which the adversary tries to distinguish could for instance be two runs of the voting scheme, with the votes of two voters being exchanged: $\mathsf{Exp}^{0}(\lambda)$ would then have Alice vote 0 and Bob 1, while $\mathsf{Exp}^{1}(\lambda)$ would have Alice vote 1 and Bob 0. A more general formalisation would let the adversary himself choose which votes are used: the adversary would propose two votes v_0, v_1, and, depending on β, Alice would vote v_0 and Bob v_1, or the other way around. Even more generally, in the formalisations we use in later chapters, the votes of more than just two voters can be swapped.

Such notions of security are typically proved by reduction: that is, by proving that breaking the security of the protocol entails breaking some security assumption on the underlying cryptographic primitives, which in turn breaks some underlying computational hardness assumption (*e.g.* RSA, factoring,...).

Computational models are much more precise than symbolic models, as they do not rely on as strong abstractions, and make much weaker assumptions on the primitives. On the other hand, the abstractions and idealisations made by symbolic models allow for much easier automated verification, as we discuss in the next section. In some cases, however, it has been proved that with reasonable assumptions symbolic models can be computationally sound, in the sense that proving symbolic security implies computational security [9, 62, 63]. Another recent approach that bridges the symbolic and computational worlds is the Bana-Comon model [19]. It considers a symbolic representation of an attacker without restricting it to a fixed set of operations, contrary to usual symbolic models, which lets the model be computationally sound by construction, while allowing for symbolic reasoning.

6 Automated verification

Symbolic vs computational Manually proving the security of a given protocol, either in symbolic or computational models, tends to be rather tedious, even for simple examples, and is prone to errors that can be difficult to detect. A natural idea is then to design procedures and tools to automatically produce these proofs. As mentioned in the previous paragraph, computational models use a much thinner abstraction layer than symbolic models, which makes automated reasoning more difficult in these models. The only significant tools that allow to reason in a computational setting are CryptoVerif [32], that automates transformations between games, EasyCrypt [21], that makes use of probabilistic relational Hoare logic, and F* [96], a functional programming language that provides a rich type system to encode security properties. Only CryptoVerif is fully automatic, while EasyCrypt and F* can rather be seen as interactive provers that assist the user in creating and verifying a proof.

We will be focusing more here on verification tools in the symbolic setting, which allows for much easier automation.

Trace and equivalence properties Symbolic verification for cryptographic protocols typically studies properties that can be classified in two categories. The first one is the class of *trace properties*, *i.e.* properties expressed as a condition that must be satisfied by all executions (also called *traces*) of the protocol. That is typically the case of reachability properties, that require that a certain "bad" state can never be reached. They can for instance express secrecy guarantees: the execution can never reach a state where the attacker has learned the value of some secret. In the context of voting protocols, verifiability is expressed as a trace property: basically, the execution should never reach a point where a result has been published that does not correctly account for all the votes.

On the other hand, some properties, called *equivalence properties*, can only be expressed by considering several executions of the protocol. Equivalence properties express the idea that two situations should be indistinguishable in the eyes of an attacker. These properties are very useful to express many types of privacy guarantees, such as, for instance, anonymity – an attacker should not be able to distinguish between me running the protocol or someone else. Equivalence can be used to formalise strong flavours of secrecy: an attacker should not only be unable to learn some secret data (*e.g.* a key), but he should even be unable to distinguish a run of the protocol that uses the actual secret from a variant that uses a different value. Unlinkability, which requires that an attacker cannot "track" a user, *i.e.* recognise whether two different sessions of a protocol are run by the same user or not, is another major example of an equivalence property. Vote privacy, as explained earlier, is also typically expressed as an equivalence property.

Automated tools were first built to verify trace properties. In this thesis, we will tend to focus more on the verification of equivalence properties, whose study is more recent, although several procedures were designed in the last few years.

Bounded vs unbounded number of sessions Symbolic models allow to consider an unbounded number of sessions of a protocol running in parallel. Verifying security properties with such an arbitrary number of sessions gives stronger guarantees: only proving them for a given fixed number of sessions might miss some attacks that require a larger number of sessions to be carried out. However, unfortunately, it has been proved that the deduction problem – can an attacker learn a given message when interacting with a certain protocol? – as well as the equivalence problem are undecidable when an unbounded number of sessions is considered [75, 52]. The procedures and tools that have been designed for automated verification then follow one of two approaches: either they restrict their scope to bounded numbers of sessions – and can then be exact decision procedures; or they consider the unbounded case, but are then incomplete – typically, they prove properties in some cases, but cannot always conclude or terminate.

Tools In the unbounded case, one of the most prominent tools is ProVerif [31, 33], which was first proposed by Blanchet in 2001, and has been steadily improved and extended since then. Basically, ProVerif is fully automatic, and proves (non-)deducibility properties by translating protocols expressed in a variant of the applied pi-calculus into a set of Horn clauses that overapproximate the knowledge an attacker can gain from interacting with the protocol. The problem is then reduced to determining whether some message is deducible from this set of Horn clauses. It has then been extended [35] to prove equivalence of processes that only differ by the messages exchanged, and not the structure of the process – *diff-equivalence* – in a similar way.

ProVerif is correct, in the sense that when it declares a protocol secure, the property indeed holds. It is however not complete: it can in some cases not terminate, or find false attacks that do not disprove the property, due to the overapproximation it performs being too coarse. For example, ProVerif's overapproximation tends to treat all processes as if they were replicated, and hence is at risk to find false attacks when the security of a protocol relies on the fact that something happens only once. This can notably be the case when considering voting protocols that require that each voters votes at most once. The main other tool for the unbounded case is Tamarin [90, 22], which uses multiset rewriting rather than process algebras and Horn clauses, but, similarly to ProVerif, handles trace properties and diff-equivalence. Tamarin however is not fully automatic, but rather interactive: it allows the user to provide lemmas to guide the tool. This allows Tamarin to prove cases that would be difficult to handle automatically, but of course requires more work from the user.

On the other hand, when restricting the study to bounded numbers of sessions, several procedures have been designed recently, that *decide* protocol equivalence. One of the first such tools was SPEC [68], which proves a strong notion of equivalence (bisimulation) for a fixed signature of cryptographic primitives. More recent tools rather prove a weaker notion, called trace equivalence. APTE [45] decides trace equivalence, also for a fixed set of cryptographic primitives, but can handle more complex constructions such as else-branches. Akiss [43] uses Horn clauses, and can handle user-provided primitives and equational theories (within a large class that captures most usual primitives). More recently, Deepsec [48] can, as Akiss, handle user-specified equational theories and complex constructions (such as else branches, private channels). It is much more efficient than previous tools – it was designed to establish an upper-bound on the complexity of the problem of deciding equivalence, as part of a proof that the problem is coNEXP-complete. An advantage of both Akiss and Deepsec is that they can take advantage of parallelisation to speed up verification. Finally, SAT-Equiv [55] can handle a fixed equational theory (that contains most usual primitives), and reduces equivalence to a graph planning problem. It is very efficient, but however imposes some restrictions on the protocols considered – it requires a well-typedness property, and as a result tags may need to be added to messages when encoding the protocol. All of these tools share the common issue that they to some extent work by considering all possible executions of the processes considered, which must be done to decide equivalence: they therefore tend to suffer, to different degrees, from efficiency issues when considering processes that involve a large numbers of parallel branches.

Typing Type systems are a common tool from the domain of Programming Languages, that are used to statically enforce safety properties such as the absence of runtime errors in programs. A simple use of types is to describe the nature of the values considered, with types such as `int`, `string`, `bool` and so on. Programs are then checked using type systems containing rules such as the following (very simple) example, that states that the product of two integers is also an integer:

$$\frac{x : \texttt{int} \qquad y : \texttt{int}}{x * y : \texttt{int}}$$

Type systems can be seen as overapproximating the possible behaviours of programs, by abstracting the values into their types. The advantage of such an approach is that it provides efficient (but of course incomplete) ways of proving properties that would be undecidable in general.

Type systems can also be used to prove much stronger properties in the context of security. They have been applied in particular to the verification of trace properties such as authentication

or secrecy properties [80, 38, 77]. In such a case, one could for instance tag the values that should not be published with a type `secret` and values that may safely be published with a type `public`. The following simple typing rule would then indicate that the encryption of a secret message does not leak any secret information, and is thus safe to publish, provided that the key used is secret:

$$\frac{x : \texttt{secret} \qquad k : \texttt{secret}}{\texttt{enc}(x, k) : \texttt{public}}$$

One would then typecheck the whole protocol to ensure that all messages that can ever be output to the network can be given type `public`.

We have already evoked F⋆ [96]. It, as well as its relational extension rF⋆ [20], relies on a rich type system with dependent and refinement types: types that contain logical formulas, which can encode complex conditions on the values they are given to. Contrary to the approach described above, its type system is not specifically dedicated to security protocols, but instead allows to manually write logical formulas, that can encode very complex properties such as cryptographic games. Typechecking thus produces complex proof obligations, that are discharged by F⋆ to a SAT-solver. Such techniques have been used to produce verified implementations of very large and complex protocols such as TLS [72]. However, a downside of such complex types is that although the typechecking itself is automated, highly non-trivial work is required from the user to come up with the correct types that will allows proofs to go through.

7 Contributions

This thesis brings several contributions to the domain of formal analysis of electronic voting protocols, on two main aspects:

- first, the automated verification of symbolic equivalence properties such as vote privacy, with the design of a type system for equivalence;
- second, a study of the notions of privacy and verifiability, that leads to the surprising result that privacy in fact implies individual verifiability, and to the design of a new game-based definition of privacy, that allows to consider a dishonest ballot box.

This section presents these contributions more in detail, and briefly describes other work performed during my PhD, that I chose not to present in this thesis.

A type system for symbolic equivalence We propose a new approach to the problem of automated proofs of equivalence properties, that makes use of type systems. As explained earlier, these have been applied to the case of trace properties such as reachability/secrecy properties. The difficulty is that such techniques typically overapproximate the set of possible executions of a protocol by abstracting the messages into their types. They can then ensure that some "bad" state (*e.g.* where the attacker learns a secret) can never be reached, by proving the stronger statement that such a state is never reached by the overapproximated set of executions. However, this would not be a sound way of doing things for equivalence. Even if the overapproximations of two processes P and Q are equivalent, it could be that some specific behaviours are actually possible in P but not in Q, which would make P and Q not equivalent, but might be missed due to the approximation. Instead, we design a type system that can typecheck two processes P and Q together, to ensure that they have the same behaviours, and we use the types to establish invariants regarding the messages stored in variables by these processes. While typechecking, our

type system collects all output messages (or in fact, subterms of these messages) into what we call a *constraint*. Some additional checks on this constraint ensure that not only P and Q have similar behaviours, but also the messages an attacker can observe in corresponding executions of the two processes are indistinguishable. Our type system handles a fixed signature of cryptographic primitives that includes most usual ones.

We prove that our type system soundly implies symbolic equivalence, even when considering processes with a mix of replicated (*i.e.* executed an unbounded number of times) and non-replicated parts.

We also implemented, in collaboration with Niklas Grimm, a prototype of our type-checking procedure, which we apply to several examples of protocols from the literature (*e.g.* key exchange protocols), in order to compare its performance to that of the other automatic tools for symbolic equivalence listed earlier. This benchmark shows that for bounded numbers of sessions, our approach is very efficient, and suffers much less than the other tools from large numbers of sessions/parallel branches. In the unbounded case, our approach can handle a mix of replicated and non-replicated processes. Our overapproximation is more precise than ProVerif's in such cases, and can leverage the fact that some part of the processes can only be executed once. This flexibility notably allows us to prove vote privacy for Helios, for which ProVerif cannot conclude. Indeed, as previously evoked, Roenne's attack implies that the property only holds if voters only vote once, which causes ProVerif to find false attacks. In exchange, our prototype requires light type annotations to be provided by hand by the user, and is less general *w.r.t.* the equational theory considered than ProVerif.

Privacy and verifiability We study the definitions of vote privacy and verifiability, and establish that, surprisingly, privacy implies individual verifiability. This result, while counter-intuitive, does not contradict the previous result from [49], as we consider established definitions with a polynomial adversary, rather than an unbounded one. Let us explain the basic intuition of our result. Consider a voting system that is not verifiable at all, *i.e.* where an adversary can somehow prevent the vote of any voter of his choice from being counted, without being detected. In such a system, an adversary trying to learn my vote could simply block all other voters' votes: the result itself would then count only my vote, and therefore reveal it. That attack would of course only work only in such an extremely non-verifiable case: we show how to generalise this intuition to produce an attack on privacy from any attack on individual verifiability. We conduct our proof in both symbolic and computational models, two very different settings, to further show that this implication is not an artefact from the chosen model, but an intrinsic property of the notions of privacy and verifiability. An important aspect of our result, that perhaps explains why it is counter-intuitive, is that it holds when considering the same trust assumptions *w.r.t.* the election authorities for both privacy and verifiability. In existing work however, verifiability is usually studied against an attacker that controls the ballot box, while privacy notions typically implicitly assume the ballot box to be trusted.

Privacy against a dishonest ballot box This last remark highlights a limitation of existing (game-based) definitions for vote privacy: they implicitly consider a trusted ballot box, while voting systems typically aim at protecting voters from a dishonest ballot box. A similar observation was made in [67, 29]. This leads to the last main contribution presented in this thesis: the design of a definition for privacy against a dishonest ballot box, in a computational setting. Our notion is based on BPRIV, a state-of-the-art notion of vote privacy against a honest ballot box, first introduced in [26]. Designing a meaningful definition when the ballot box is

untrusted is made noticeably difficult by the attack described in the previous paragraph. An adversary that controls the ballot box can always remove ballots from it, which leads to the result giving out more information than it should about the remaining ones. This trivially violates the naïve privacy notion that requires that no matter what, the adversary cannot gain information about the votes. We present a way of defining a slightly weaker privacy notion, which does not consider the behaviour above an attack, but still provide meaningful guarantees. More precisely, we show that our notion is stil sufficiently robust, in the sense that it implies a simulation-based notion of vote privacy. This simulation-based property states that the voting system securely implements an ideal functionality, which makes clear what power an adversary can have against the system. We then apply our privacy notion to a few voting protocols from the literature, namely Helios [10], Belenios [59] and Civitas [54], to illustrate that it allows for a more precise characterisation of the level of privacy they offer.

BeleniosVS During my PhD, I also worked on the design and proof of the BeleniosVS voting protocol [56]. BeleniosVS is an extension of Belenios [59] and BeleniosRF [44], which makes use of Voting Sheets, *i.e.* sheets containing pre-encrypted ballots that are distributed to voters. Voters cast their vote by scanning the ballot of their choice from their sheet on the device they use to participate in the election, rather than by having that device encrypt the vote itself. This way, the device never has access to the vote in clear, which allows the system to guarantee vote privacy to voters without requiring them to trust their device. I participated in proving that BeleniosVS guarantees privacy, verifiability, and receipt-freeness properties. I chose not to present this work in this thesis.

QUIC Finally, an internship at Microsoft Research in Cambridge gave me the opportunity to participate in the verification of the QUIC protocol. QUIC is a new protocol originally proposed by Google and currently being standardised by IETF. It provides a similar functionality as TLS, but relies on the network protocol UDP, while TLS relies on TCP. In particular, this allows QUIC to receive and decrypt data packets out of order, without having to wait for previous packets to arrive, which allows for lower latency and better performance. I participated in developing a security model of the QUIC packet construction in F*, and discovering a slight vulnerability in the way some header data in the packets are protected. This work is still under development, and will also not be presented in this thesis.

8 Outline

The thesis is organised into two parts of three chapters each, the first being dedicated to the type system for protocol equivalence, and the second to the study of privacy and verifiability definitions.

- Chapter 1 introduces the rather standard symbolic model we use, and the formal definition of the trace equivalence notion we consider.
- Chapter 2 presents the type system for equivalence as well as the attached procedure for constraints, and the proof of their soundness for both bounded and unbounded numbers of sessions.
- In Chapter 3 we conduct a case study by detailing the application of our type system to two relevant examples of protocols (Helios and Private Authentication). We also present the experimental results of our prototype implementation.

- Chapter 4 presents our result that vote privacy implies individual verifiability in a symbolic model.
- Chapter 5 details the same result in a computational model.
- In Chapter 6, finally, we expose our new definition for vote privacy against a dishonest ballot box, prove that it implies a simulation-based notion of privacy, and study its properties, notably applying it to real examples of voting protocols.

Related publications The work on the type system for equivalence detailed in Part I has been published in two articles [60] and [61], presented respectively at the conferences CCS'17 (ACM Conference on Computer and Communications Security) and POST'18 (Principles of Security and Trust). These two articles were written in collaboration with Véronique Cortier, Niklas Grimm and Matteo Maffei.

The work on privacy and verifiability (Chapters 4 and 5) has been published in an article [64] presented at the CCS'18 conference.

The work on BeleniosVS, not detailed in this thesis, was published in an article [56] presented at the CSF'19 (IEEE symposium on Computer Security Foundations).

In addition, the source code for the implementation of the typechecker, the example files for the benchmark, as well as the scripts used to generate the graphs featured in Chapter 6 are available at [1].

Part I

A Type System for Equivalence

Introduction to Part I

Formal methods proved to be indispensable tools for the analysis of advanced cryptographic protocols. As explained in introduction, mature push-button analysis tools have emerged and have been successfully applied to many protocols from the literature in the context of *trace properties* such as authentication or confidentiality. A current and very active topic is the adaptation of these techniques to the more involved case of *trace equivalence* properties. These are the natural symbolic counterpart of cryptographic indistinguishability properties, and they are at the heart of many privacy properties.

For instance, consider an electronic voting protocol, such as Helios evoked in introduction, involving a part $Voter(id, v)$ run by voter id who wishes to vote for candidate v, and a part representing the server's role S. This scheme preserves secrecy of the vote if an attacker cannot know who voted for which candidate. That is, the attacker should not be able to distinguish an execution where Alice votes for 0 and Bob for 1 from an execution where Alice votes for 1 and Bob for 0. This can be expressed by the equivalence

$$Voter(Alice, 0) \,|\, Voter(Bob, 1) \,|\, S \;\approx_t\; Voter(Alice, 1) \,|\, Voter(Bob, 0) \,|\, S.$$

Another example of an application of equivalence properties is unlinkability. Consider for instance the case of RFID protocols, that typically involve an RFID tag and a reader, and aim to let the tag authenticate the reader. These are for instance used by electronic passports, or contactless cards for public transportation. A basic requirement of such protocols is that they should correctly ensure authentication. They should also preserve the secrecy of data stored on the tag (*e.g.* keys, ...). These requirements are trace properties. However, they do not cover all security guarantees one might want for RFID tags. A potential issue of these protocols is that, if not designed carefully, they might allow an attacker to track a user, *i.e.* to recognise and identify them by simply interacting with their tag. To prevent such privacy breaches, a protocol must satisfy *unlinkability*: an attacker should not be able to link different sessions of the protocol to the same user. Consider a protocol $Reader(k) \,|\, Tag(k)$, in which a reader is communicating with a tag with which it shares a secret key k. Unlinkability can be expressed (adapting the formalisation proposed in [13]) as the equivalence

$$\begin{array}{l} Reader(k_{Alice}) \mid Tag(k_{Alice}) \mid Reader(k_{Alice}) \mid Tag(k_{Alice}) \;\approx_t \\ Reader(k_{Alice}) \mid Tag(k_{Alice}) \mid Reader(k_{Bob}) \;\mid Tag(k_{Bob}) \end{array}$$

requiring that the attacker cannot determine whether he sees the same tag (belonging to Alice) twice, or two different tags (belonging to Alice and Bob respectively).

In addition to vote privacy and unlinkability, trace equivalence is used to express many privacy properties, such as differential privacy [76], or anonymity [7, 14].

Related work As we detailed in introduction, several tools and procedures have been proposed in recent years to automatically verify equivalence properties in the symbolic model. As the problem is undecidable when considering an unbounded number of sessions of protocols [52], these tools use one of two approaches. Either they restrict themselves to bounded numbers of sessions only, in which case the problem becomes decidable, or they handle the unbounded case, at the cost of being incomplete – typically relying on overapproximations and heuristics to prove equivalence, with no termination guarantees.

Tools that decide equivalence in the bounded case notably encompass SPEC [68], APTE [45, 18], Akiss [43], and more recently Deepsec [48] and SAT-Equiv [55]. These tools vary in the class of cryptographic primitives and of protocols they can consider. However, as they *decide* the problem of equivalence, due to its complexity – it is coNEXP-complete [48] – they all tend to suffer from the state explosion problem. Most of the older tools can typically not analyse more than a few sessions of even relatively small protocols. The more recent tools – SAT-Equiv and Deepsec – are more efficient and depending on the protocol considered may go a bit further, but they still inevitably suffer when considering large numbers of sessions.

The only tools that can verify equivalence properties for an unbounded number of sessions are ProVerif [35], Maude-NPA [95], and Tamarin [22]. ProVerif checks a property that is stronger than trace equivalence, namely diff-equivalence, which works well in practice provided that protocols have a similar structure. However, as for trace properties, the internal design of ProVerif, that relies on translating the protocols to a set of Horn clauses, renders the tool unable to distinguish between exactly one session and infinitely many. This overapproximation often yields false attacks, in particular when the security of a protocol relies on the fact that some action is only performed once. Maude-NPA also checks diff-equivalence but often does not terminate. Tamarin, based on multiset rewriting, can handle an unbounded number of sessions and is very flexible in terms of supported protocol classes but it often requires human interactions.

Contribution In this part, we consider a novel type checking-based approach. Several sound type systems have successfully been developed for proving trace properties (*e.g.* [80, 38, 39]). One advantage is that a few typing annotations often suffice to convey the reasons why a protocol is secure. Intuitively, a type system over-approximates protocol behaviour. Due to this over-approximation, it is no longer possible to *decide* security properties but the types typically convey sufficient information to *prove* security. Extending this approach to equivalence properties is a delicate task. Indeed, two protocols P and Q are in equivalence if (roughly) any trace of P has an equivalent trace in Q (and conversely). Overapproximating behaviours may not preserve equivalence.

Instead, we develop a somewhat hybrid approach: we design a type system to overapproximate the set of possible traces and we collect the set of sent messages into *constraints*. We then propose a procedure for proving (static) equivalence of the constraints. These do not only contain sent messages but also reflect internal checks made by the protocols, which is crucial to guarantee that whenever a message is accepted by P, it is also accepted by Q (and conversely).

After introducing our symbolic model, as well as the relevant notions and properties in Chapter 1, we present our type system in detail in Chapter 2. We show that our type system soundly proves equivalence of protocols for both a bounded and an unbounded number of sessions, or a mix of both. This is particularly convenient to analyse systems where some actions are limited (*e.g.*, no revote, or limited access to some resource). More specifically, we show that whenever two protocols P and Q are type-checked to be equivalent, then they are in trace equivalence, for the standard notion of trace equivalence [37], against a full Dolev-Yao attacker.

In particular, one advantage of our approach is that it proves security directly in a security model that is similar to the ones used by the other popular tools, in contrast to many other security proofs based on type systems. Our result holds for protocols with all standard primitives (symmetric and asymmetric encryption, signatures, pairs, hash), with atomic keys (that can be fresh or long-term keys) and no private channels. Similarly to ProVerif, we need the two protocols P and Q to have a rather similar structure.

In Chapter 3, we present a case study, where we use our type system to prove vote privacy for Helios, and anonymity for the Private Authentication protocol. In addition, in collaboration with Niklas Grimm, we designed a prototype implementation of our type system. We evaluate its performance on several protocols of the literature. In the case of a bounded number of sessions, our tool provides a significant speed-up compared to most other tools. To be fair, let us emphasise that these tools can *decide* equivalence while our tool checks sufficient conditions by the means of our type system. In the case of an unbounded number of sessions, the performance of our prototype tool is comparable to ProVerif. In contrast to ProVerif, our tool can consider a mix of bounded and unbounded number of sessions. As an application, we can prove for the first time ballot privacy of the well-known Helios e-voting protocol [10], without assuming a reliable channel between honest voters and the ballot box. ProVerif fails in this case as ballot privacy only holds under the assumption that honest voters vote at most once, otherwise the protocol is subject to a copy attack [94]. For similar reasons, Tamarin also fails to verify this protocol.

Related publications The three chapters in this part are based on two articles [60] and [61], published respectively at the conferences CCS'17 (ACM Conference on Computer and Communications Security) and POST'18 (Principles of Security and Trust). These two articles were written in collaboration with Véronique Cortier, Niklas Grimm and Matteo Maffei.

Chapter 1

Preliminaries: Symbolic Model and Pi-calculus

1.1 Introduction

In this chapter, we present the syntax and semantics of the symbolic model we consider. Typically, in such models, cryptographic operations are idealised and represented by abstract function symbols. Messages are then constructed using these abstract cryptographic primitives. The abilities of the attacker, *i.e.* what he can compute given a set of messages using the properties of the primitives, are modelled as a rewriting system on messages. Security protocols are then modelled as processes of a process algebra, such as the applied pi-calculus [6]. We present here a calculus close to [47] inspired from the calculus underlying the ProVerif tool [34].

1.2 Messages and Terms

Messages are modelled as terms built over a signature of abstract cryptographic primitives.

Formally, we consider an infinite set of names $\mathcal{N}$ for nonces, which are used to model random values generated during an execution of the protocol. We also assume an infinite set of variable names $\mathcal{V}$.

Cryptographic primitives are modelled through a *signature* $\mathcal{F}$, that is, a set of function symbols, given with their arity (*i.e.* the number of arguments).

Given a signature $\mathcal{F}$, a set of names $\mathcal{N}$, and a set of variables $\mathcal{V}$, the set of *terms* $\mathcal{T}(\mathcal{F},\mathcal{V},\mathcal{N})$ is the set inductively defined by applying functions to variables in $\mathcal{V}$ and names in $\mathcal{N}$. That is, the smallest set that

- contains $\mathcal{V}$ and $\mathcal{N}$: $\mathcal{V}\cup\mathcal{N}\subseteq\mathcal{T}(\mathcal{F},\mathcal{V},\mathcal{N})$;
- and is closed under application of function symbols in $\mathcal{F}$:

$$\forall f\in\mathcal{F} \text{ with arity } n.\ \forall t_1,\ldots,t_n\in\mathcal{T}(\mathcal{F},\mathcal{V},\mathcal{N}).\ f(t_1,\ldots,t_n)\in\mathcal{T}(\mathcal{F},\mathcal{V},\mathcal{N}).$$

A *substitution* $\sigma=\{M_1/x_1,\ldots,M_k/x_k\}$ is a mapping from variables $x_1,\ldots,x_k\in\mathcal{V}$ to messages $M_1,\ldots,M_k$. We let $\mathrm{dom}(\sigma)=\{x_1,\ldots,x_k\}$. We say that σ is ground if all messages $M_1,\ldots,M_k$ are ground. We let $\mathrm{names}(\sigma)=\bigcup_{1\leq i\leq k}\mathrm{names}(M_i)$. The application of a substitution σ to a term t is denoted $t\sigma$. Intuitively, applying σ to a term t is the operation of replacing

each occurrence of a variable $x \in \mathrm{dom}(\sigma)$ in t with the message $\sigma(x)$. Formally, this operation is defined inductively as follows.

$$\forall n \in \mathcal{N}.\ n\sigma = n$$
$$\forall x \in \mathcal{V}.\ x\sigma = \begin{cases} \sigma(x) & \text{if } x \in \mathrm{dom}(\sigma) \\ x & \text{otherwise} \end{cases}$$
$$\forall f \in \mathcal{F} \text{ with arity } n.\ \forall t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V}, \mathcal{N}).\ f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$$

In the following, we will consider the signature:

$$\mathcal{F}_c = \{\mathtt{pk}/1, \mathtt{vk}/1, \mathtt{enc}/2, \mathtt{aenc}/2, \mathtt{sign}/2, \langle\cdot,\cdot\rangle/2, \mathtt{h}/2\}$$

whose function symbols model respectively public and verification keys, symmetric and asymmetric encryption, concatenation and hash. The companion primitives (symmetric and asymmetric decryption, signature check, and projections) are represented by the following signature:

$$\mathcal{F}_d = \{\mathtt{dec}/2, \mathtt{adec}/2, \mathtt{checksign}/2, \pi_1/1, \pi_2/1\}$$

We also consider a set $\mathcal{C}$ of public constants, *i.e.* function symbols of arity 0, used as agent names for instance.

We assume the set $\mathcal{N}$ of names is partitioned into the set $\mathcal{FN}$ of free nonces, intended to represent nonces created by the attacker, and the set $\mathcal{BN}$ of bound nonces, used to represent nonces created by the protocol parties. We similarly consider an infinite set of names $\mathcal{K}$ for keys, split into sets $\mathcal{FK}$ and $\mathcal{BK}$, that represent keys generated respectively by the attacker, and by protocol parties. Finally we assume the set of variables to be split into two subsets $\mathcal{V} = \mathcal{X} \uplus \mathcal{AX}$ where $\mathcal{X}$ are variables used in processes while $\mathcal{AX}$ are variables used by the attacker to store messages.

For any term t, we denote by $\mathrm{names}(t)$ (resp. $\mathrm{keys}(t)$, $\mathrm{vars}(t)$) the set of names (resp. keys, variables) occurring in t. A term is *ground* if it does not contain variables.

We consider the set $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$ of *cryptographic terms*, simply called *terms*. *Messages* are terms containing only constructors, *i.e.* from $\mathcal{T}(\mathcal{F}_c \cup \mathcal{C}, \mathcal{V}, \mathcal{N} \cup \mathcal{K})$.

An *attacker term*, is a term from $\mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}, \mathcal{AX}, \mathcal{FN} \cup \mathcal{FK})$. In particular, an attacker term cannot use nonces and keys created by the protocol's parties.

Example 1. *For instance, the following terms M and N are messages that represent the encryption of respectively a name $n \in \mathcal{N}$ and a variable $x \in \mathcal{X}$ with a key $k \in \mathcal{BK}$:*

$$M = \mathtt{enc}(n, k), \quad N = \mathtt{enc}(x, k).$$

M is ground, while N is not. We have $\mathrm{names}(M) = \{n\}$, $\mathrm{names}(N) = \emptyset$, *and* $\mathrm{keys}(M) = \mathrm{keys}(N) = \{k\}$.

The following term R is an attacker term that represents the attacker attempting to decrypt a message he has stored in a variable $y \in \mathcal{AX}$ with a key stored in a variable $z \in \mathcal{AX}$:

$$R = \mathtt{dec}(y, z)$$

Note that attacker terms cannot contain keys in $\mathcal{BK}$: intuitively the attacker does not know a priori any key used by the protocol agents. To decrypt a message with a key, as in R, he thus needs to have obtained and stored that key in one of his variables.

The *evaluation* of a term t, denoted $t{\downarrow}$, corresponds to the bottom-up application of the cryptographic primitives. Basically, destructors are applied when possible: *e.g.* decrypting an encrypted message with the correct key returns the plaintext. For instance, $\mathtt{dec}(\mathtt{enc}(0,k),k)$, where k is a key, evaluates to 0. When a destructor is present but cannot be applied, the evaluation fails. For instance, trying to decrypt a ciphertext with the wrong key fails: $\mathtt{dec}(\mathtt{enc}(0,k),k')$, where $k \neq k'$ are two different keys, fails and evaluates to $\bot$. In addition, evaluation enforces that the keys used are always atomic, *i.e.* elements of $\mathcal{K}$, rather than constructed from other messages. Evaluating a message where a non-atomic key is used, *e.g.* $\mathtt{enc}(0,\langle k,k'\rangle)$ where a pair composed of two keys is used as an encryption key, will fail, and yield $\bot$.

Formally, $t{\downarrow}$ for a term t is recursively defined as follows.

$$
\begin{array}{rl}
u{\downarrow} = u & \text{if } u \in \mathcal{N} \cup \mathcal{V} \cup \mathcal{K} \cup \mathcal{C} \\
\mathtt{pk}(t){\downarrow} = \mathtt{pk}(t{\downarrow}) & \text{if } t{\downarrow} \in \mathcal{K} \\
\mathtt{vk}(t){\downarrow} = \mathtt{vk}(t{\downarrow}) & \text{if } t{\downarrow} \in \mathcal{K} \\
\mathtt{h}(t){\downarrow} = \mathtt{h}(t{\downarrow}) & \text{if } t{\downarrow} \neq \bot \\
\langle t_1,t_2\rangle{\downarrow} = \langle t_1{\downarrow}, t_2{\downarrow}\rangle & \text{if } t_1{\downarrow} \neq \bot \text{ and } t_2{\downarrow} \neq \bot \\
\mathtt{enc}(t_1,t_2){\downarrow} = \mathtt{enc}(t_1{\downarrow}, t_2{\downarrow}) & \text{if } t_1{\downarrow} \neq \bot \text{ and } t_2{\downarrow} \in \mathcal{K} \\
\mathtt{sign}(t_1,t_2){\downarrow} = \mathtt{sign}(t_1{\downarrow}, t_2{\downarrow}) & \text{if } t_1{\downarrow} \neq \bot \text{ and } t_2{\downarrow} \in \mathcal{K} \\
\mathtt{aenc}(t_1,t_2){\downarrow} = \mathtt{aenc}(t_1{\downarrow}, t_2{\downarrow}) & \text{if } t_1{\downarrow} \neq \bot \text{ and } t_2{\downarrow} = \mathtt{pk}(k) \text{ for some } k \in \mathcal{K} \\
\pi_1(t){\downarrow} = t_1 & \text{if } t{\downarrow} = \langle t_1,t_2\rangle \\
\pi_2(t){\downarrow} = t_2 & \text{if } t{\downarrow} = \langle t_1,t_2\rangle \\
\mathtt{dec}(t_1,t_2){\downarrow} = t_3 & \text{if } t_1{\downarrow} = \mathtt{enc}(t_3,t_4) \text{ and } t_4 = t_2{\downarrow} \\
\mathtt{adec}(t_1,t_2){\downarrow} = t_3 & \text{if } t_1{\downarrow} = \mathtt{aenc}(t_3,\mathtt{pk}(t_4)) \text{ and } t_4 = t_2{\downarrow} \\
\mathtt{checksign}(t_1,t_2){\downarrow} = t_3 & \text{if } t_1{\downarrow} = \mathtt{sign}(t_3,t_4) \text{ and } t_2{\downarrow} = \mathtt{vk}(t_4) \\
t{\downarrow} = \bot & \text{otherwise.}
\end{array}
$$

Note in particular that this evaluation operation models encryption algorithms that are length-hiding: the only computation that can be performed on a ciphertext is to decrypt it with the decryption key. Hence, from a ciphertext, an adversary cannot recover any information on the length of the plaintext. Additionally, the asymmetric encryption operation described by the model is anonymous, in the sense that a ciphertext reveals no information on the public key used to produce it.

Also note that the evaluation of term t succeeds only if all underlying keys are atomic, even when these keys would disappear from the final message during evaluation. For example, while $\mathtt{dec}(\mathtt{enc}(0,k),k){\downarrow} = 0$, we have $\mathtt{dec}(\mathtt{enc}(0,\langle k,k\rangle),\langle k,k\rangle){\downarrow} = \bot$, because the message $\langle k,k\rangle$ is not an atomic key, even though it disappears during the evaluation.

We write $t =_{\downarrow} t'$ if $t{\downarrow} = t'{\downarrow}$.

1.3 Processes and Semantics

Security protocols describe how messages should be exchanged between participants. We model them through a process algebra, whose syntax is displayed in Fig. 1.1.

We identify processes up to α-renaming, which is defined as usual. To avoid capture of bound variables and names, we then assume that all bound names, keys, and variables in the process have been renamed so that they are all are distinct.

A *configuration* of the system is a tuple $(\mathcal{P};\phi;\sigma)$ where:

- $\mathcal{P}$ is a multiset of processes that represents the current active processes.

Destructors used in processes:

$$d ::= \mathtt{dec}(x,t) \mid \mathtt{adec}(x,t) \mid \mathtt{checksign}(x,t') \mid \pi_1(x) \mid \pi_2(x)$$

where $x \in \mathcal{X}$, $t \in \mathcal{K} \cup \mathcal{X}$, $t' \in \{\mathtt{vk}(k) | k \in \mathcal{K}\} \cup \mathcal{X}$.

Processes:

$$\begin{aligned} P, Q ::= \;& 0 \\ \mid \;& \mathtt{new}\ n.P \\ \mid \;& \mathtt{out}(M).P \\ \mid \;& \mathtt{in}(x).P \\ \mid \;& (P \mid Q) \\ \mid \;& \mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q \\ \mid \;& \mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q \\ \mid \;& !P \end{aligned}$$

where $n \in \mathcal{BN} \cup \mathcal{BK}$, $x \in \mathcal{X}$, and M, N are messages.

Figure 1.1 – Syntax for processes.

- ϕ is a substitution with $\mathrm{dom}(\phi) \subseteq \mathcal{AX}$ and for any $x \in \mathrm{dom}(\phi)$, $\phi(x)$ (also denoted $x\phi$) is a message that only contains variables in $\mathrm{dom}(\sigma)$. ϕ represents the terms that have been sent on the network by the processes.
- σ is a ground substitution, that contains the messages with which variables in the processes are instantiated in the current execution.

The semantics of processes is given through a transition relation $\xrightarrow{\alpha}$ on configurations, defined in Figure 1.2. α is the *action* associated to the transition. This action represents what an attacker can observe when the transition is executed. It can be

- τ, which denotes a silent action, *i.e.* a transition where the attacker cannot observe anything;
- $\mathtt{in}(R)$, which denotes that the attacker provides as input to the process a message constructed by applying an attacker term R, called the *recipe*, to the frame of messages previously output;
- or $\mathtt{new}\ ax.\mathtt{out}(ax)$, which denotes that a message is output to the attacker by the process, and stored in variable ax in the frame.

The relation $\xrightarrow{w}_*$ is defined as the reflexive transitive closure of $\xrightarrow{\alpha}$, where w is the concatenation of all actions. We also write equality up to silent actions $=_\tau$.

Intuitively, process $\mathtt{new}\ n.P$ creates a fresh nonce or key, and behaves like P. Process $\mathtt{out}(M).P$ emits M and behaves like P, provided that the evaluation of M is successful. The corresponding message is stored in the substitution ϕ, corresponding to the attacker knowledge. The attacker controls the communication between processes, in the sense that a process may input any message that the attacker can forge (rule In). More precisely, the attacker uses a recipe R to compute a new message from his knowledge ϕ. Note that all names, except the attacker names $\mathcal{FN}$, are initially assumed to be secret, and similarly for keys. Process $P \mid Q$ corresponds to the parallel composition of P and Q. Process $\mathtt{let}\ x = d\ \mathtt{in}\ P\ \mathtt{else}\ Q$ behaves like P in which x is replaced by d if d can be successfully evaluated, and behaves like Q otherwise.

$$(\{P_1 \mid P_2\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P_1, P_2\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{Par}$$

$$(\{0\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\mathcal{P}; \phi; \sigma) \quad \textsc{Zero}$$

$$(\{\texttt{new}\ n.P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{New}$$

$$(\{\texttt{new}\ k.P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{NewKey}$$

$$(\{\texttt{out}(t).P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\texttt{new}\ ax_n.\texttt{out}(ax_n)} (\{P\} \cup \mathcal{P}; \phi \cup \{t/ax_n\}; \sigma) \quad \textsc{Out}$$
if $t\sigma$ is a ground term, $(t\sigma)\!\downarrow \neq \bot$, $ax_n \in \mathcal{AX}$ and $n = |\phi| + 1$

$$(\{\texttt{in}(x).P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\texttt{in}(R)} (\{P\} \cup \mathcal{P}; \phi; \sigma \cup \{(R\phi\sigma)\!\downarrow / x\}) \quad \textsc{In}$$
if R is an attacker term such that $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi)$, and $(R\phi\sigma)\!\downarrow \neq \bot$

$$(\{\texttt{let}\ x = d\ \texttt{in}\ P\ \texttt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma \cup \{(d\sigma)\!\downarrow / x\}) \quad \textsc{Let-In}$$
if $d\sigma$ is ground and $(d\sigma)\!\downarrow \neq \bot$

$$(\{\texttt{let}\ x = d\ \texttt{in}\ P\ \texttt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{Let-Else}$$
if $d\sigma$ is ground and $(d\sigma)\!\downarrow = \bot$, *i.e.* d fails

$$(\{\texttt{if}\ M = N\ \texttt{then}\ P\ \texttt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{If-Then}$$
if M, N are messages such that $M\sigma$, $N\sigma$ are ground, $(M\sigma)\!\downarrow \neq \bot$, $(N\sigma)\!\downarrow \neq \bot$, and $M\sigma = N\sigma$

$$(\{\texttt{if}\ M = N\ \texttt{then}\ P\ \texttt{else}\ Q\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{Q\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{If-Else}$$
if M, N are messages such that $M\sigma$, $N\sigma$ are ground and $(M\sigma)\!\downarrow = \bot$ or $(N\sigma)\!\downarrow = \bot$ or $M\sigma \neq N\sigma$

$$(\{!P\} \cup \mathcal{P}; \phi; \sigma) \xrightarrow{\tau} (\{P, !P\} \cup \mathcal{P}; \phi; \sigma) \quad \textsc{Repl}$$

Figure 1.2 – Semantics

Process $\texttt{if}\ M = N\ \texttt{then}\ P\ \texttt{else}\ Q$ behaves like P if M and N correspond to two equal messages and behaves like Q otherwise. The replicated process $!P$ behaves as an unbounded number of copies of P in parallel. In the following, we will often omit the else-branch in conditionals and destructor applications when it is empty, *i.e.* 0. Similarly, we will omit the null process at the end of branches, *e.g.* simply writing $\texttt{in}(x).\texttt{out}(M)$ for $\texttt{in}(x).\texttt{out}(M).0$.

A *trace* of a process P is any possible sequence of transitions in the presence of an attacker that may read, forge, and send messages. Formally, the set of traces $\mathrm{trace}(P)$ is defined as follows.

$$\mathrm{trace}(P) = \{(w, \phi, \sigma) | (\{P\}; \emptyset; \emptyset) \xrightarrow{w}_* (\mathcal{P}; \phi; \sigma)\}$$

Example 2. *Consider the Helios protocol mentioned in introduction. Informally, the behaviour of the protocol is as follows:*

$$V_i \to S : \quad [\{v_i\}^{r_i}_{\texttt{pk}(k_s)}]_{k_i}$$
$$S \to V_1, \ldots, V_n : \ v_1, \ldots, v_n$$

where $\{m\}^r_{pk}$ denotes the asymmetric encryption of message m with public key pk randomised with r; and $[m]_k$ denotes the signature of m with key k. Voters $V_1, \ldots, V_n$ each send their vote v_i encrypted with the public key $\texttt{pk}(k_s)$ of the election to the server S on an authentic channel (modelled here by a signature). For simplicity, compared to the informal description from the introduction, we have combined here the roles of the ballot box and the tallying authority into a single entity S. The server decrypts all ballots, to compute and publish the result, i.e. *the set of votes in clear.*

We describe here a simplified version of Helios, incorporating the weeding operation, in a simple case with only two (honest) voters A and B and a voting server S. The protocol can be

modelled by the process:

$$H = \texttt{new}\ r_a.\mathit{Voter}(k_a, v_a, r_a) \ \mid\ \texttt{new}\ r_b.\mathit{Voter}(k_b, v_b, r_b) \ \mid\ P_S$$

where $\mathit{Voter}(k, v, r)$ *represents voter* k *willing to vote for* v *using randomness* r *while* P_S *represents the voting server.* $\mathit{Voter}(k, v, r)$ *encrypts the vote* v*, using the value* r *to randomise the encryption; and signs the ciphertext before outputting it, which models sending the ciphertext over an authenticated channel.*

$$\mathit{Voter}(k, v, r) = \texttt{out}(\texttt{sign}(\texttt{aenc}(\langle v, r\rangle, \texttt{pk}(k_S)), k))$$

Note that we model the randomised asymmetric encryption by having the random value r *appear paired together with the payload* v*. This of course an abstraction, which does not reflect how the randomness would be used in the actual construction of the cryptographic primitive. At the level of abstraction of our symbolic model however, this correctly models the behaviour of randomised encryption.*

The voting server receives ballots from A *and* B *and then outputs the decrypted ballots, after some mixing.*

$$\begin{aligned} P_S = {} & \texttt{in}(x_1).\texttt{in}(x_2). \\ & \texttt{let}\ y_1 = \texttt{checksign}(x_1, \texttt{vk}(k_a))\ \texttt{in} \\ & \texttt{let}\ y_2 = \texttt{checksign}(x_2, \texttt{vk}(k_b))\ \texttt{in} \\ & \texttt{let}\ z_1 = \texttt{adec}(y_1, k_s)\ \texttt{in} \quad \texttt{let}\ z_1' = \pi_1(z_1)\ \texttt{in} \\ & \texttt{let}\ z_2 = \texttt{adec}(y_2, k_s)\ \texttt{in} \quad \texttt{let}\ z_2' = \pi_1(z_2)\ \texttt{in} \\ & \quad (\texttt{out}(z_1') \mid \texttt{out}(z_2')) \end{aligned}$$

Let us describe a possible execution of process H*, that describes a "normal" run of the protocol, where the attacker does not interfere. We can first apply rule* Par *twice to* H*, to split the parallel branches. Then rules* New *and* Out *can be applied to process* $\texttt{new}\ r_a.\mathit{Voter}(k_a, v_a, r_a)$*, storing message* $\texttt{sign}(\texttt{aenc}(\langle v_a, r_a\rangle, \texttt{pk}(k_S)), k_a)$ *into the variable* $ax_1 \in \mathcal{AX}$*. The same rules can be applied to* $\texttt{new}\ r_b.\mathit{Voter}(k_b, v_b, r_b)$*, storing* $\texttt{sign}(\texttt{aenc}(\langle v_b, r_b\rangle, \texttt{pk}(k_S)), k_b)$ *into variable* $ax_2 \in \mathcal{AX}$*.*

We can then apply rule In *twice to process* P_S*, inputting the two messages stored in* ax_1*,* ax_2*. Consequently these two messages are stored in variables* x_1, x_2*. The messages have the expected form: applying rule* Let-In *six times hence successfully applies all destructors, storing all intermediary results in variables* y_1, y_2, z_1, z_2*, and finally messages* v_a*,* v_b *into* z_1' *and* z_2'*. Rule* Par *can then be applied to split the two parallel branches* $\texttt{out}(z_1')$*,* $\texttt{out}(z_2')$*. We may then apply rule* Out *to any of the two, say the first one:* v_a *is stored in variable* ax_3*, and then the other:* v_b *is stored in* ax_4*.*

At this point, there are no more processes to be reduced, except for implicit null processes: rule Zero *takes care of these, and the execution reaches its end.*

Altogether, the execution can be summarised by

$$(\{H\}, \emptyset, \emptyset) \xrightarrow{t}_* (\emptyset, \phi, \sigma)$$

where the sequence t *of actions (ignoring silent actions) is*

$t =_\tau \texttt{new}\ ax_1.\texttt{out}(ax_1).\ \texttt{new}\ ax_2.\texttt{out}(ax_2).\ \texttt{in}(ax_1).\ \texttt{in}(ax_2).\ \texttt{new}\ ax_3.\texttt{out}(ax_3).\ \texttt{new}\ ax_4.\texttt{out}(ax_4)$,

the substitution ϕ contains all output messages:

$$\begin{aligned}\phi = [&ax_1 \mapsto \mathtt{sign}(\mathtt{aenc}(\langle v_a, r_a\rangle, \mathtt{pk}(k_S)), k_a),\\ &ax_2 \mapsto \mathtt{sign}(\mathtt{aenc}(\langle v_b, r_b\rangle, \mathtt{pk}(k_S)), k_b),\\ &ax_3 \mapsto z'_1,\ ax_4 \mapsto z'_2],\end{aligned}$$

and finally the substitution σ stores the contents of all variables in the processes:

$$\begin{aligned}\sigma = [&x_1 \mapsto \mathtt{sign}(\mathtt{aenc}(\langle v_a, r_a\rangle, \mathtt{pk}(k_S)), k_a),\\ &x_2 \mapsto \mathtt{sign}(\mathtt{aenc}(\langle v_b, r_b\rangle, \mathtt{pk}(k_S)), k_b),\\ &y_1 \mapsto \mathtt{aenc}(\langle v_a, r_a\rangle, \mathtt{pk}(k_S)),\\ &y_2 \mapsto \mathtt{aenc}(\langle v_b, r_b\rangle, \mathtt{pk}(k_S)),\\ &z_1 \mapsto \langle v_a, r_a\rangle, z_2 \mapsto \langle v_b, r_b\rangle,\\ &z'_1 \mapsto v_a, z'_2 \mapsto v_b].\end{aligned}$$

Note that the execution of this process is not deterministic. For instance we could have chosen, when reducing process P_S, to first perform the output of z'_2 and then that of z'_1. This would have produced the reduction

$$(\{H\}, \emptyset, \emptyset) \xrightarrow{t}_* (\emptyset, \phi', \sigma),$$

with the same actions t and substitution σ as the execution above, but a different substitution ϕ', that is similar to ϕ, but with the contents of variables ax_3 and ax_4 being swapped.

1.4 Equivalence

When processes evolve, sent messages are stored in a substitution ϕ while the values of variables are stored in σ. A *frame* is simply a substitution ψ where $\mathrm{dom}(\psi) \subseteq \mathcal{AX}$. It represents the knowledge of an attacker. In what follows, we will typically consider $\phi\sigma$.

Intuitively, two sequences of messages are indistinguishable to an attacker if he cannot perform any test that could distinguish them. This is typically modelled as static equivalence [6]. Here, we consider of variant of [6] where the attacker is also given the ability to observe when the evaluation of a term fails, as defined for example in [47].

Definition 1 (Static Equivalence)**.** *Two ground frames ϕ and ϕ' are statically equivalent if and only if they have the same domain, and for all attacker terms R, S with variables in $\mathrm{dom}(\phi) = \mathrm{dom}(\phi')$, we have*

$$(R\phi =_\downarrow S\phi) \iff (R\phi' =_\downarrow S\phi').$$

We then write $\phi \approx \phi'$.

Example 3. *Consider constant values $a, b \in \mathcal{C}$, and keys $k, k' \in \mathcal{K}$. The following static equivalence holds:*

$$[ax_1 \mapsto \mathtt{enc}(a, k)] \ \approx\ [ax_1 \mapsto \mathtt{enc}(b, k)].$$

Indeed, even though two different public values a, b are used on either side, they are encrypted with a key k, and the frame does not reveal this key. Thus the attacker is unable to observe this difference. If however the frame reveals key k, the equivalence is broken:

$$[ax_1 \mapsto \mathtt{enc}(a, k),\ ax_2 \mapsto k] \ \not\approx\ [ax_1 \mapsto \mathtt{enc}(b, k),\ ax_2 \mapsto k].$$

Indeed, the attacker can now use k to decrypt and observe a difference. For instance, using recipes $R = \mathtt{dec}(ax_1, ax_2)$ and $S = a$, we have $R\phi = a = S\phi$, while $R\phi' = b \neq S\phi'$ (where ϕ, ϕ' are the frames on the left and on the right).

Then two processes P and Q are in equivalence if no matter how the adversary interacts with P, a similar interaction may happen with Q, with equivalent resulting frames.

Definition 2 (Trace Equivalence)**.** *Let P, Q be two processes. We write $P \sqsubseteq_t Q$ if for all $(s, \phi, \sigma) \in \text{trace}(P)$, there exists $(s', \phi', \sigma') \in \text{trace}(Q)$ such that $s =_\tau s'$ and $\phi\sigma$ and $\phi'\sigma'$ are statically equivalent. We say that P and Q are trace equivalent, and we write $P \approx_t Q$, if $P \sqsubseteq_t Q$ and $Q \sqsubseteq_t P$.*

Note that this definition already includes the attacker's behaviour, since processes may input any message forged by the attacker.

Example 4. *For instance, the following trace equivalence holds:*

$$\texttt{new}\ k.\texttt{out}(\texttt{enc}(a,k)) \ \approx_t\ \texttt{new}\ k.\texttt{out}(\texttt{enc}(b,k)).$$

Indeed, these two processes can only produce (with the same sequences of actions) the two frames $[ax_1 \mapsto \texttt{out}(\texttt{enc}(a,k))]$ and $[ax_2 \mapsto \texttt{out}(\texttt{enc}(b,k))]$, which are statically equivalent as explained in the previous example.

On the other hand, outputting the key k breaks the equivalence:

$$\texttt{new}\ k.\texttt{out}(\texttt{enc}(a,k)).\texttt{out}(k) \ \not\approx_t\ \texttt{new}\ k.\texttt{out}(\texttt{enc}(b,k)).\texttt{out}(k).$$

Indeed, the process on the left can produce the frame $[ax_1 \mapsto \texttt{enc}(a,k),\ ax_2 \mapsto k]$, and the process on the right (with the same actions) can only produce the frame $[ax_1 \mapsto \texttt{enc}(b,k),\ ax_2 \mapsto k]$. As explained in the previous example, these two frames are not statically equivalent: hence, there exists a behaviour of the left process that cannot be mimicked by any behaviour of the right process. Thus, $P \not\sqsubseteq_t Q$, which implies $P \not\approx_t Q$.

As evoked in introduction, ballot privacy is typically modelled as an equivalence property [71] that requires that an attacker cannot distinguish when Alice is voting 0 and Bob is voting 1 from the scenario where the two votes are swapped.

Continuing Example 2, ballot privacy of Helios can be expressed as follows:

$$\begin{aligned} &\texttt{new}\ r_a.Voter(k_a, 0, r_a) \ \mid\ \texttt{new}\ r_b.Voter(k_b, 1, r_b) \ \mid\ P_S \\ \approx_t\ &\texttt{new}\ r_a.Voter(k_a, 1, r_a) \ \mid\ \texttt{new}\ r_b.Voter(k_b, 0, r_b) \ \mid\ P_S \end{aligned}$$

This notion of equivalence is the property our type system entails. In the next chapter, we present the type system in detail, and establish that it is sound, *i.e.* that it correctly implies trace equivalence.

Chapter 2

Type System

2.1 Introduction

As explained in introduction, type systems are a powerful tool to statically verify safety properties in the context of programming languages, but can also be used to enforce much stronger security guarantees in the case of cryptographic protocols. The advantage of typing-based approaches is that they typically produce efficient, although by design incomplete, ways to prove properties that are undecidable in general. As mentioned earlier, they have been applied in the context of security protocols to prove trace properties such as authentication and secrecy properties (*e.g.* [80, 38, 39]). Typically, this kind of approach consists in using types to carry information (*e.g.* labels indicating the level of confidentiality) about the messages exchanged. This basically forms an overapproximation of all possible executions of a protocol, by considering the types as an abstraction of the messages. It is then possible to check that this overapproximated set of executions never violates the trace property considered, *e.g.* by publishing a message whose type indicates it should be secret. However, such an overapproximation would not be sound when considering equivalence properties. Indeed, even if the overapproximations of the possible traces of two processes P and Q are equivalent, it does not mean that the actual processes are. It might be that some behaviour of P is actually impossible in Q, but that the overapproximation misses that fact. Such a case would break the equivalence of P and Q. Instead, we propose an approach where the two processes are typechecked at the same time, with typing rules designed to ensure that they have the same behaviours. As we will see, this is not sufficient: we also need to check that the messages exchanged by the processes on the left and on the right are also equivalent. To that end, our typechecking rules collect these messages, and some additional conditions are checked on them afterwards, to ensure they do not leak information.

This chapter presents the type system we propose to statically verify trace equivalence properties for pi-calculus processes, as defined in the previous chapter. We then establish the soundness of the type system for protocols with a bounded number of sessions, *i.e.* without replication, and finally lift this result to the case of unbounded numbers of sessions.

2.2 Overview

We first introduce the key ideas underlying our approach on (the fixed version of) the Helios voting protocol [10]. Helios is a verifiable voting protocol, that we have described informally in introduction. It that has been used in various elections, including the election of the rector of

the University of Louvain-la-Neuve. We described a pi-calculus process modelling a simplified version of Helios in Chapter 1. We consider here a slightly more detailed model, that we will now present. The informal behaviour of the system, with a server S and n voters $V_1, \ldots, V_n$, is depicted below.

$$\begin{aligned} S &\to V_i : && r_i \\ V_i &\to S : && [\{v_i\}^{r_i, r'_i}_{\mathtt{pk}(k_s)}]_{k_i} \\ S &\to V_1, \ldots, V_n : && v_1, \ldots, v_n \end{aligned}$$

where $\{m\}^r_{\mathtt{pk}(k)}$ denotes the asymmetric encryption of message m with the key $\mathtt{pk}(k)$ randomised with the nonce r, and $[m]_k$ denotes the signature of m with key k. v_i is a value in the set $\{0, 1\}$, which represents the candidate that voter V_i votes for. In the first step, the voter casts her vote, encrypted with the election's public key $\mathtt{pk}(k_s)$ and then signed. Since generating a good random number is difficult for the voter's client (typically a JavaScript run in a browser), a typical trick is to input some randomness (r_i) from the server and to add it to its own randomness (r'_i). In the second step the server outputs the tally (*i.e.*, a randomised permutation of the valid votes received in the voting phase). Note that the original Helios protocol does not assume signed ballots. Instead, voters authenticate themselves through a login mechanism. For simplicity, we abstract this authenticated channel by a signature.

As in the model from the previous chapter, we consider here an abstraction of the tallying process from the actual Helios protocol. In the real scheme, the voting server only collects ballots, and a different entity, the tallying authority, is in charge of computing and publishing the result. Moreover this tallying entity is actually split into several trustees who each have only a share of the decryption key, to distribute the trust among them. We consider here the case of a honest voting server and tallying authority – although the attacker is still able to block and intercept the ballots sent by voters to the server. Hence, for simplicity, we conflate the server and talliers into a single entity, that retrieves all ballots and publishes the result of the election.

As already explained in introduction, a voting protocol provides vote privacy [71] if an attacker is not able to know which voter voted for which candidate. Intuitively, this can be modelled as the following trace equivalence property, which requires the attacker not to be able to distinguish A voting 0 and B voting 1 from A voting 1 and B voting 0.

$$\begin{aligned} & \mathit{Voter}(k_a, 0) \mid \mathit{Voter}(k_b, 1) \mid \mathit{CompromisedVoters} \mid S \\ \approx_t \; & \mathit{Voter}(k_a, 1) \mid \mathit{Voter}(k_b, 0) \mid \mathit{CompromisedVoters} \mid S \end{aligned}$$

where $\mathit{Voter}(k, v)$ is a process modelling a voter with private key k voting for candidate v, S is a process modelling the behaviour of the server, and *CompromisedVoters* a process representing voters controlled by the attacker. These would typically simply give their keys to the attacker, and let him vote for them. The attacker may a priori control an unbounded number of voters.

Despite its simplicity, this protocol has a few interesting features that make its analysis particularly challenging. First of all, as discussed in introduction, the server is supposed to discard ciphertext duplicates, otherwise a malicious eligible voter E could intercept A's ciphertext, sign it, and send it to the server [65], as exemplified below:

$$\begin{aligned} A &\to S : && [\{v_a\}^{r_a, r'_a}_{\mathtt{pk}(k_s)}]_{k_a} \\ E &\to S : && [\{v_a\}^{r_a, r'_a}_{\mathtt{pk}(k_s)}]_{k_e} \\ B &\to S : && [\{v_b\}^{r_b, r'_b}_{\mathtt{pk}(k_s)}]_{k_b} \\ S &\to A, B : && v_a, v_b, v_a \end{aligned}$$

This would make the two tallied results distinguishable, thereby breaking trace equivalence, since $v_a, v_b, v_a \not\approx_t v_a, v_b, v_b$.

Even more interestingly, each voter is supposed to be able to vote *only once*, otherwise the same attack would apply [94] even if the server discards ciphertext duplicates (as the randomness used by the voter in the two ballots would be different). This makes the analysis particularly challenging, and in particular out of scope of existing cryptographic protocol analysers such as ProVerif. Indeed, as mentioned earlier, ProVerif abstracts away from the number of protocol sessions, and due to this abstraction reports this attack even when it is made impossible by having voters vote only once.

In the following sections, we introduce a type system to statically check trace equivalence between processes. Our typing judgements thus capture properties of pairs of terms or processes, which we will refer to as the *left* and *right* term or process, respectively. Typing judgments are parametrised by a *typing environment* Γ, which is a mapping from names, keys, and variables to types. We will usually denote typing environments Γ.

Let us now give an intuition of what types we need to typecheck the aforementioned equivalence property. We first assume standard security labels: `HH` stands for high confidentiality and high integrity, `HL` for high confidentiality and low integrity (*e.g.* a message encrypted with a public key), and `LL` for low confidentiality and low integrity.[1]

In the case of Helios, the nonces r_a and r_b are sent by the server in clear and unauthenticated: we will give them label `LL`. However, the nonces r'_a and r'_b generated by the voters should not be published, hence we will give them label `HH`.

More precisely, we will use type $\tau_i^{l,1}$ to describe randomness of security label l produced by the randomness generator at position i in the program, which can be invoked at most once. As we consider processes up to α-renaming, we will assume that each distinct invocation of the randomness generator in the process uses a different name (*e.g.* r_a, r_b, etc. in the Helios example). We will then use this name to refer to the position i of the corresponding call to the generator. For instance, nonce r'_a will thus be given type $\tau_{r'_a}^{\texttt{HH},1}$.

We will also use a type $\tau_i^{l,\infty}$, which is similar, except that the randomness generator can be invoked an unbounded number of times, *i.e.* the corresponding "`new`" instruction occurs in a replicated process. Although this "infinite" type does not occur in the case of Helios, since we purposefully prevent revote, it will be useful later on when considering other examples. These types induce a partition on random values, in which each set contains at most one element or an unbounded number of elements, respectively. This turns out to be useful, as explained below, to type-check protocols, like Helios, in which the number of times messages of a certain shape are produced is relevant for the security of the protocol.

We also give security labels to keys: k_a, k_b and k_s should remain secret, and we will thus give them label `HH`. Let us emphasise that these labels are not assumptions, but rather an indication to guide the typechecking. That is, we do not *assume* that these keys are secret, but rather we indicate that they *should* not be published, and the typechecking will ensure that they are indeed not.

The security label is not sufficient when reasoning about keys: we also want, at least when a key is trusted, to somehow encode information regarding what messages it can encrypt or sign. To do this, the complete type we use for keys has the form $\mathsf{key}^l(T)$, where l is the label of the

[1] We omit the low confidentiality and high integrity type, which could be added but was not necessary to typecheck any of the protocols we considered.

key itself, and T the type of the messages it can encrypt or sign. When the key is trusted (*i.e.*, $l = \mathtt{HH}$), the type system will ensure that it correctly only encrypts messages of the right type.

In the case of Helios, the protocol specifies that k_a should only be used to sign encryptions of the vote of A, together with the randomness r_a, r'_a. To express this, we use a type $\{T\}_{k_s}$, that describes the asymmetric encryption of a message of type T with the key k_s[2]. As we will see later on, we will need to encode precisely in the payload type T what is the value of this vote. To do this we introduce a refinement type $[\![\tau_0^{\mathtt{LL},1}\,;\,\tau_1^{\mathtt{LL},1}]\!]$, that describes with the same notations as the types for nonces the value – here 0 or 1 – of the term, both on the left and on the right. All in all, putting all these ingredients together, we give the following type to k_a:

$$k_a : \mathrm{key}^{\mathtt{HH}}(\{[\![\tau_0^{\mathtt{LL},1}\,;\,\tau_1^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r'_a}^{\mathtt{HH},1}\}_{k_s}).$$

A similar type conveys the information that k_b is a trusted key, that signs only the encryption of Bob's vote, *i.e.* 1 on the left and 0 on the right:

$$k_b : \mathrm{key}^{\mathtt{HH}}(\{[\![\tau_1^{\mathtt{LL},1}\,;\,\tau_0^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r'_b}^{\mathtt{HH},1}\}_{k_s}).$$

The type of k_s is similar: it also has label $\mathtt{HH}$ since it is trusted. However, this time there are several possibilities for the messages it encrypts. Indeed, it can be used to encrypt Alice's vote, or Bob's vote: thus, it inherits the two payload types, which are combined in disjunctive form. Since the associated encryption key is public – it is the election public key – it can also be used by the attacker to encrypt a dishonest vote, or in fact any arbitrary message. To reflect that, public key types implicitly convey an additional payload type, the one characterising messages encrypted by the attacker: these are of low confidentiality and integrity. Following these observations, we can give the following type to k_s:

$$k_s : \mathrm{key}^{\mathtt{HH}}(([\![\tau_0^{\mathtt{LL},1}\,;\,\tau_1^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r'_a}^{\mathtt{HH},1}) \;\vee\; ([\![\tau_1^{\mathtt{LL},1}\,;\,\tau_0^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r'_b}^{\mathtt{HH},1})).$$

Key types are crucial to type-check the server code: we verify the signatures produced by A and B and then derive from the types of k_a and k_b the types for these signed messages. We can then use this ciphertext type to infer after decryption the vote cast by A and B, respectively. While processing the other ballots, to avoid the replay attack described above, the server discards any additional ciphertexts that are equal to the ones from A and B. To use the information obtained from the types of the randomness, we encode this behaviour by having server discard ciphertexts produced with randomness matching the one used by A or B. Given that these random values are used only once, we know that the remaining ciphertexts must come from the attacker and thus convey the same vote on the left- and on the right-hand side. This suffices to type-check the final output of the votes in clear: the votes from Alice and Bob, as a set, are $\{0, 1\}$ on both sides – as derived from the refinements in the ciphertext types – and other votes are dishonest and thus the same on both sides. Thus the two tallied results on the left- and right-hand side are the same, which fulfils trace equivalence.

The information carried by the types, and analysed during the typechecking, is however not sufficient in general for equivalence. Consider for instance a key k with a simple type $\mathrm{key}^{\mathtt{HH}}(\mathtt{HL})$, *i.e.* a trusted key (label $\mathtt{HH}$) that can encrypt any message ($\mathtt{HL}$ being the most generic type: the one for messages that should not be published, but cannot be trusted either). Consider then the processes $P = \mathtt{out}(\mathtt{enc}(0,k)).\mathtt{out}(\mathtt{enc}(0,k))$, and $Q = \mathtt{out}(\mathtt{enc}(0,k)).\mathtt{out}(\mathtt{enc}(1,k))$.

[2] In the actual types later on, we will actually only specify the type of the key, rather than the key itself, in the encryption type. We write the key itself in this overview for clarity.

P outputs twice the same public value 0, encrypted with k, while Q first encrypts 0 and then 1. This is perfectly fine *w.r.t.* the type of the key k: that trusted key is allowed to encrypt any message. Note however that we did not (contrary to the Helios example) randomise the encryption here: hence the two ciphertexts published by P are the same, while the two messages from Q are different. This trivially makes P and Q non-equivalent, but cannot be detected by the key type. Indeed, the key type only enforces local conditions each time the key is used to encrypt or decrypt, while the problem here is more global – it involves different messages, in different outputs, that could be in different parts of a real protocol. To detect such issues, we need some sort of global view of the messages exchanged.

To produce this global view, the type system generates a set containing the sequence of all pairs of messages produced in the outputs of the left and right processes – or in fact, as we will explain in later sections, subterms of these messages. We call this set a *constraint*. For instance, just to give an idea of what a constraint looks like, the constraints generated in the Helios example are reported below:

$$\begin{aligned} C = \{(\{&\mathtt{sign}(\mathtt{aenc}(\langle 0, \langle x, r'_a\rangle\rangle, \mathtt{pk}(k_S)), k_a) \sim \mathtt{sign}(\mathtt{aenc}(\langle 1, \langle x, r'_a\rangle\rangle, \mathtt{pk}(k_S)), k_a),\\ &\mathtt{aenc}(\langle 0, \langle x, r'_a\rangle\rangle, \mathtt{pk}(k_S)) \sim \mathtt{aenc}(\langle 1, \langle x, r'_a\rangle\rangle, \mathtt{pk}(k_S)),\\ &\mathtt{sign}(\mathtt{aenc}(\langle 1, \langle y, r'_b\rangle\rangle, \mathtt{pk}(k_S)), k_b) \sim \mathtt{sign}(\mathtt{aenc}(\langle 0, \langle y, r'_b\rangle\rangle, \mathtt{pk}(k_S)), k_b),\\ &\mathtt{aenc}(\langle 1, \langle y, r'_b\rangle\rangle, \mathtt{pk}(k_S)) \sim \mathtt{aenc}(\langle 0, \langle y, r'_b\rangle\rangle, \mathtt{pk}(k_S))\},\\ &[x : \mathtt{LL}, y : \mathtt{LL}])\} \end{aligned}$$

Once typechecking is done, we basically need to check for the kind of repetitions displayed above in the constraints. Intuitively, we need the sequences of messages on the left and the right to be statically equivalent, which is the case in the Helios example above. The actual general condition is actually a bit more complex, as these messages may contain uninstantiated variables, for which we want to use the type information collected during typechecking. We will later on define formally that condition, called *consistency*, that characterises the indistinguishability of the messages output by the process. We are then able to show that typechecking with consistent constraints soundly implies trace equivalence.

Non-uniformity A central assumption that would simplify the analysis when checking for equivalence is uniform execution, meaning that the two processes of interest always take the same branch in a branching statement. For instance, this means that all decryptions must always succeed or fail equally in the two processes. That is for instance the case in the example described above. It would be possible to enforce this condition, by restricting the type system so that encryption and decryption are only allowed with keys whose equality on the left and on the right can be statically proved.

There are however protocols that require non-uniform execution for a proof of trace equivalence, *e.g.*, the private authentication protocol [7]. This protocol aims at authenticating B to A, anonymously *w.r.t.* other agents. More specifically, agent B may refuse to communicate with agent A but a third agent D should not learn whether B declines communication with A or not. The protocol can be informally described as follows, where $\mathtt{pk}(k)$ denotes the public key associated to key k, and $\mathtt{aenc}(M, \mathtt{pk}(k))$ denotes the asymmetric encryption of message M with this public key.

$$\begin{aligned} A \to B :&\quad \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, \mathtt{pk}(k_b))\\ B \to A :&\quad \begin{cases} \mathtt{aenc}(\langle N_a, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) & \text{if } B \text{ accepts } A\text{'s request}\\ \mathtt{aenc}(N_b, \mathtt{pk}(k)) & \text{if } B \text{ declines } A\text{'s request} \end{cases} \end{aligned}$$

$\Gamma(k_b, k_b)$	=	$\text{key}^{\texttt{HH}}(\texttt{HL} * \texttt{LL})$	initial message uses same key on both sides
$\Gamma(k_a, k)$	=	$\text{key}^{\texttt{HH}}(\texttt{HL})$	authentication succeeded on the left, failed on the right
$\Gamma(k, k_c)$	=	$\text{key}^{\texttt{HH}}(\texttt{HL})$	authentication succeeded on the right, failed on the left
$\Gamma(k_a, k_c)$	=	$\text{key}^{\texttt{HH}}(\texttt{HL})$	authentication succeeded on both sides
$\Gamma(k, k)$	=	$\text{key}^{\texttt{HH}}(\texttt{HL})$	authentication failed on both sides

Figure 2.1 – Key types for the private authentication protocol

If B declines to communicate with A, he sends a decoy message $\texttt{aenc}(N_b, \texttt{pk}(k))$ where $\texttt{pk}(k)$ is a decoy key (no one knows the private key k).

Encrypting with different keys Let $P_a(k_a, \texttt{pk}(k_b))$ model agent A willing to talk with B, and $P_b(k_b, \texttt{pk}(k_a))$ model agent B willing to talk with A (and declining requests from other agents). We model the protocol as:

$$
\begin{array}{rcl}
P_a(k_a, pk_b) & = & \texttt{new}\ N_a.\texttt{out}(\texttt{aenc}(\langle N_a, \texttt{pk}(k_a)\rangle, pk_b)).\ \texttt{in}(z) \\
P_b(k_b, pk_a) & = & \texttt{new}\ N_b.\ \texttt{in}(x). \\
 & & \texttt{let}\ y = \texttt{adec}(x, k_b)\ \texttt{in}\ \ \texttt{let}\ y_1 = \pi_1(y)\ \texttt{in}\ \ \texttt{let}\ y_2 = \pi_2(y)\ \texttt{in} \\
 & & \quad \texttt{if}\ y_2 = pk_a\ \texttt{then} \\
 & & \quad\quad \texttt{out}(\texttt{aenc}(\langle y_1, \langle N_b, \texttt{pk}(k_b)\rangle\rangle, pk_a)) \\
 & & \quad \texttt{else}\ \texttt{out}(\texttt{aenc}(N_b, \texttt{pk}(k)))
\end{array}
$$

where $\texttt{adec}(M, k)$ denotes asymmetric decryption of message M with private key k. We model anonymity as the following equivalence, intuitively stating that an attacker should not be able to tell whether B accepts requests from the agent A or C:

$$P_a(k_a, \texttt{pk}(k_b)) \ \mid\ P_b(k_b, \texttt{pk}(k_a)) \ \approx_t\ P_a(k_a, \texttt{pk}(k_b)) \ \mid\ P_b(k_b, \texttt{pk}(k_c))$$

We now show how we can type the protocol in order to show trace equivalence. The initiator P_a is trivially executing uniformly, since it does not contain any branching operations. We hence focus on typing the responder P_b.

The beginning of the responder protocol can be typed using standard techniques. Then however, we perform the test $y_2 = \texttt{pk}(k_a)$ on the left side and $y_2 = \texttt{pk}(k_c)$ on the right side. Since we cannot statically determine the result of the two equality checks – and thus guarantee uniform execution – we have to typecheck the four possible combinations of `then` and `else` branches. This means we have to typecheck outputs of encryptions that use different keys on the left and the right side.

To deal with this we do not assign types to single keys, but rather to pairs of keys (k, k') – which we call *bikeys* – where k is the key used in the left process and k' is the key used in the right process. The key types used for typing are presented in Fig. 2.1.

As an example, we consider the combination of the `then` branch on the left with the `else` branch on the right. This combination occurs when A is successfully authenticated on the left side, while being rejected on the right side. We then have to typecheck B's positive answer together with the decoy message: $\Gamma \vdash \texttt{aenc}(\langle y_1, \langle N_b, \texttt{pk}(k_b)\rangle\rangle, \texttt{pk}(k_a)) \sim \texttt{aenc}(N_b, \texttt{pk}(k)) : \texttt{LL}$. For this we need the type for the bikey (k_a, k).

$$
\begin{array}{lcl}
l & ::= & \texttt{LL} \mid \texttt{HL} \mid \texttt{HH} \\
KT & ::= & \text{key}^{l}(T) \mid \text{eqkey}^{l}(T) \mid \text{seskey}^{l,a}(T) \text{ with } a \in \{1, \infty\} \\
T & ::= & l \\
& \mid & T * T \\
& \mid & T \vee T \\
& \mid & KT \mid \text{pkey}(KT) \mid \text{vkey}(KT) \\
& \mid & (T)_T \mid \{T\}_T \\
& \mid & [\![\tau_n^{l,a} ; \tau_m^{l',a}]\!] \text{ with } a \in \{1, \infty\}
\end{array}
$$

Figure 2.2 – Types for terms

Decrypting non-uniformly When decrypting a ciphertext that was potentially generated using two different keys on the left and the right side, we have to take all possibilities into account. Consider the following extension of the process P_a where agent A decrypts B's message.

$$
\begin{array}{rcl}
P_a(k_a, pk_b) & = & \texttt{new}\ N_a.\texttt{out}(\texttt{aenc}(\langle N_a, \texttt{pk}(k_a)\rangle, pk_b)).\ \texttt{in}(z). \\
& & \texttt{let}\ z' = \texttt{adec}(z, k_a)\ \texttt{in}\ \texttt{out}(1) \\
& & \texttt{else}\ \texttt{out}(0)
\end{array}
$$

In the decryption, there are the following possible cases:

- The message is a valid encryption supplied by the attacker (using the public key $\texttt{pk}(k_a)$), so we check the `then` branch on both sides with $\Gamma(z') = \texttt{LL}$.
- The message is not a valid encryption supplied by the attacker so we check the `else` branch on both sides.
- The message is a valid response from B. The keys used on the left and the right are then one of the four possible combinations $(k_a, k), (k_a, k_c), (k, k_c)$ and (k, k).
  - In the first two cases the decryption will succeed on the left and fail on the right. We hence check the `then` branch on the left with $\Gamma(z') = \texttt{HL}$ with the `else` branch on the right. If the type $\Gamma(k_a, k)$ were different from $\Gamma(k_a, k_c)$, we would check this combination twice, using the two different payload types.
  - In the remaining two cases the decryption will fail on both sides. We hence would have to check the two `else` branches (which however we already did).

While checking the `then` branch together with the `else` branch, we have to check $\Gamma \vdash 1 \sim 0 : \texttt{LL}$, which rightly fails, as the modified protocol no longer guarantees trace equivalence.

2.3 Typing

2.3.1 Types

In our type system we give types to pairs of messages – one from the left process and one from the right one. We store the types of nonces, variables, and keys in a typing environment Γ. While we store a type for a single nonce or variable occurring in both processes, we assign a potentially different type to every different combination of keys (k, k') used in the left and right process – so called *bikeys*. This is an important non-standard feature that enables us to type protocols using different encryption and decryption keys.

$$\frac{}{T <: T}\ (\textsc{SRefl}) \qquad \frac{}{T <: \mathtt{HL}}\ (\textsc{SHigh}) \qquad \frac{T <: T' \qquad T' <: T''}{T <: T''}\ (\textsc{STrans})$$

$$\frac{}{\mathtt{LL} * \mathtt{LL} <: \mathtt{LL}}\ (\textsc{SPairL}) \qquad \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}\ (\textsc{SPair})$$

$$\frac{}{\mathtt{HH} * T <: \mathtt{HH}}\ (\textsc{SPairS}) \qquad \frac{}{T * \mathtt{HH} <: \mathtt{HH}}\ (\textsc{SPairS'})$$

$$\frac{}{\mathrm{key}^l(T) <: l}\ (\textsc{SKey}) \qquad \frac{}{\mathrm{eqkey}^l(T) <: \mathrm{key}^l(T)}\ (\textsc{SEqKey}) \qquad \frac{}{\mathrm{seskey}^{l,a}(T) <: \mathrm{eqkey}^l(T)}\ (\textsc{SSesKey})$$

$$\frac{T <: \mathrm{eqkey}^l(T')}{\mathrm{pkey}(T) <: \mathtt{LL}}\ (\textsc{SPubKey}) \qquad \frac{T <: \mathrm{eqkey}^l(T')}{\mathrm{vkey}(T) <: \mathtt{LL}}\ (\textsc{SVKey})$$

$$\frac{T <: T'}{(T)_{T''} <: (T')_{T''}}\ (\textsc{SEnc}) \qquad \frac{T <: T'}{\{T\}_{T''} <: \{T'\}_{T''}}\ (\textsc{SAEnc})$$

Figure 2.3 – Subtyping Rules

The types for messages are defined in Figure 2.2. We consider a subtyping relation $<:$ on types, which is defined by subtyping rules given in Figure 2.3. In particular, rules SRefl and STrans make it reflexive and transitive, as expected from a subtyping relation. We explain below the types and their meaning, as well as the remaining subtyping rules.

We assume three security labels $\mathtt{HH}, \mathtt{HL}$ and $\mathtt{LL}$, ranged over by l, whose first (resp. second) component denotes the confidentiality (resp. integrity) level. Intuitively, values of high confidentiality may never be output to the network in plain, and values of high integrity are guaranteed not to originate from the attacker. Type $\mathtt{HL}$ is the least informative of these levels, as it gives no information regarding the message it applies to. Such a message has high confidentiality and low integrity: it can therefore not be published, but we do not know where it originates from or what it contains. For this reason, any (typable) message could be given this type: rule SHigh expresses that any type is a subtype of $\mathtt{HL}$.

Pair types $T * T'$ describe the type of their components: such a type indicates a pair whose first component has type T and second component type T'. Four subtyping rules apply to pair types. Rule SPair is the simplest: it allows to subtype types T_1 and T_2 separately into T_1', T_2', to get that the pair type $T_1 * T_2$ is a subtype of $T_1' * T_2'$. Rule SPairL states that $\mathtt{LL} * \mathtt{LL}$ is a subtype of $\mathtt{LL}$: intuitively, it means that as long as two messages have low confidentiality and can be published, then the same is true for the pair of these two messages. Finally, rules SPairS and SPairS$'$ state that any pair type is a subtype of $\mathtt{HH}$, as long as one of its two components is $\mathtt{HH}$. In terms of messages, this corresponds to the fact that as soon as one component of the pair has high integrity, *i.e.* cannot have been constructed by the attacker, then the whole pair cannot either.

The union type $T \vee T'$ is given to messages that can have type T or type T'. We call *branches* all the types obtained by splitting the $\vee$ type. More precisely:

Definition 3 (Branches of a type)**.** *If T is a type, we write* branches(T) *the set of all types T' such that T' is not a union type, and either*

- $T = T'$*;*
- *or there exist types $T_1,\ldots,T_k,T'_1,\ldots,T'_{k'}$ such that*

$$T = T_1 \vee \ldots \vee T_k \vee T \vee T'_1 \vee \ldots \vee T'_{k'}$$

This naturally extends to typing environments as follows.

Definition 4 (Branches of an environment)**.** *For a typing environment Γ, we write* branches(Γ) *the sets of all environments Γ' such that*

- $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma)$
- $\forall x \in \mathrm{dom}(\Gamma).\quad \Gamma'(x) \in \mathrm{branches}(\Gamma(x))$.

The type $\tau_n^{l,a}$ describes nonces and constants of security level l: the label a ranges over $\{\infty, 1\}$, denoting whether the nonce is bound within a replication or not (constants are always typed with $a = 1$). We assume a different identifier n for each constant and restriction in the process. The type $\tau_n^{l,1}$ is populated by a single name, (*i.e.*, n describes a constant or a non-replicated nonce) and $\tau_n^{l,\infty}$ is a special type, that is instantiated to $\tau_{n_j}^{l,1}$ in the jth replication of the process.

Type $[\![\tau_n^{l,a}\,;\,\tau_m^{l',a}]\!]$ is a refinement type that restricts the set of possible values of a message to values of type $\tau_n^{l,a}$ on the left and type $\tau_m^{l',a}$ on the right. For a refinement type $[\![\tau_n^{l,a}\,;\,\tau_n^{l,a}]\!]$ with equal types on both sides we simply write $\tau_n^{l,a}$.

Keys can have three different types ranged over by KT, ordered by a subtyping relation (rules SEqKey, SSesKey): $\mathrm{seskey}^{l,a}(T) <: \mathrm{eqkey}^{l}(T) <: \mathrm{key}^{l}(T)$. For all three types, l denotes the security label of the key: as rule SKey indicates, all of these types are subtypes of l. T is the type of the payload that can be encrypted or signed with these keys. This allows us to transfer typing information from one process to another one: *e.g.* when encrypting, we check that the payload type is respected, so that we can be sure to get a value of the payload type upon decryption. The three different types encode different relations between the left and the right component of a bikey (k, k'). While type $\mathrm{key}^{l}(T)$ can be given to bikeys with different components $k \neq k'$, type $\mathrm{eqkey}^{l}(T)$ can only be given to a bikey provided the keys are equal on both sides. Type $\mathrm{seskey}^{l,a}(T)$ additionally guarantees that the key in question is never associated to a different key in another bikey, at any point in the execution. More precisely, type $\mathrm{eqkey}^{l}(T)$ could be given to a bikey (k, k) even if at some point the bikey (k, k') for some $k' \neq k$ occurs (this bikey (k, k') would then be given some other type $\mathrm{key}^{l'}(T')$). In contrast, if (k, k) is given type $\mathrm{seskey}^{l,a}(T)$, then k can never be associated with a different key k': a bikey with type $\mathrm{seskey}^{l,a}(T)$ is always the same on the left and the right throughout the whole process. We allow for dynamic generation of keys of type $\mathrm{seskey}^{l,a}(T)$, and use a label a to denote whether the key is generated under replication or not – just like for nonce types.

We denote the sets of all keys in an environment Γ by:

$$\mathrm{keys}(\Gamma) = \{k \in \mathcal{K} \mid \exists k' \in \mathcal{K}.\ (k, k') \in \mathrm{dom}(\Gamma) \vee (k', k) \in \mathrm{dom}(\Gamma)\}.$$

We also denote the set of all session keys in Γ by:

$$\mathrm{seskeys}(\Gamma) = \{k \in \mathcal{K} \mid \exists l, a, T.\ \Gamma(k, k) = \mathrm{seskey}^{l,a}(T)\}.$$

For a key of type T, we use types $\mathrm{pkey}(T)$ and $\mathrm{vkey}(T)$ for the corresponding public key and verification key, and types $(T')_T$ and $\{T'\}_T$ for symmetric and asymmetric encryptions of messages of type T' with this key. Public keys and verification keys can be treated as LL if the corresponding keys are equal (rules SPUBKEY,SVKEY): a public key is safe to output, provided that *the same* public key is output on the left and on the right. Subtyping on encryptions is directly induced by subtyping of the payload types (rules SENC, SAENC).

2.3.2 Constraints

When typing messages, we generate constraints of the form $(M \sim N)$, with the meaning that the attacker may see M and N in the left and right process, respectively, and that these two messages are thus required to be indistinguishable.

Definition 5 (Constraint)**.** *Formally, a constraint is defined as a pair of messages, separated by the symbol* $\sim$*:*

$$u \sim v$$

As we will see, several constraints may be generated when typing messages: we thus also consider sets of constraints, which we usually denote c. We will also consider couples (c, Γ) composed of such a set, and a typing environment Γ, that contains type information on the variables, gathered while typing the messages. Finally, when typechecking processes, we will obtain several such tuples, and will thus need to handle sets of these, which we will call *constraint sets* and usually denote C.

The next section describes the typing rules of our type system. Before we get to that, let us define a few operations on constraints and constraint sets that will be useful. We first give an intuition of these operations, before defining them formally.

When typechecking processes, as we will see in the next sections, we will obtain several constraint sets, and we will need to join them by computing what we call their *product union*. Basically, given two constraint sets C, C', this operation yields a constraint set $C \cup_\times C'$ that contains all elements of the form $c \cup c'$, where c comes from C and c' from C'. Of course, by definition of a constraint sets, c does not occur alone in C, but rather as a component of a tuple (c, Γ) (and similarly for c'). We thus also define a notion of union for typing environments. This union corresponds to the intuition of simply putting together all the types contained in the environments: the union of $\Gamma = [x : T]$ and $\Gamma' = [y : T']$ would simply be $\Gamma \cup \Gamma' = [x : T, y : T']$. This is well defined when Γ and Γ' have disjoint domains. In case they do not, say if both Γ and Γ' contain a binding for some variable z, we require that they agree on the type of z. This property, which we call *compatibility*, is formally defined below.

Definition 6 (Compatible environments)**.** *We say that two typing environments* Γ*,* Γ' *are compatible if they are equal on the intersection of their domains,* i.e. *if*

$$\forall x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma').\ \Gamma(x) = \Gamma'(x).$$

We can then define formally the union of two typing environments.

Definition 7 (Union of environments)**.** *Let* Γ*,* Γ' *be two compatible environments. Their union* $\Gamma \cup \Gamma'$ *is defined by*

- $\mathrm{dom}(\Gamma \cup \Gamma') = \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma')$
- $\forall x \in \mathrm{dom}(\Gamma).\ (\Gamma \cup \Gamma')(x) = \Gamma(x)$

- $\forall x \in \mathrm{dom}(\Gamma').\ (\Gamma \cup \Gamma')(x) = \Gamma'(x)$

Note that this function is well defined since Γ *and* Γ' *are assumed to be compatible.*

Finally we can define the following two operations on constraint sets.

Definition 8 (Operations on constraint sets)**.** *We define two operations on constraints.*

- *the product union of constraint sets:*

$$C \cup_{\times} C' := \{(c \cup c', \Gamma \cup \Gamma') \mid (c, \Gamma) \in C \wedge (c', \Gamma') \in C' \wedge \Gamma, \Gamma' \textit{ are compatible}\}$$

- *the addition of a set of constraints* c' *to all elements of a constraint set* C*:*

$$\begin{aligned} C \cup_{\forall} c' &:= C \cup_{\times} \{(c', \emptyset)\} \\ &= \{(c \cup c', \Gamma) \mid (c, \Gamma) \in C\} \end{aligned}$$

2.3.3 Typing messages

The typing judgement for messages is of the form $\Gamma \vdash M \sim N : T \to c$ which reads as follows: under the environment Γ, M and N are of type T. Either this is a high confidentiality type, *i.e.* M and N are not disclosed to the attacker, or M and N are indistinguishable for the attacker provided the set of constraints c is consistent (the proper definition of consistency is provided in a later section).

The most common use case for our typing rules for messages is when a message is output on the network: in that case we want to ensure the messages on the left and right processes can be given type LL, to check they do not let the attacker distinguish between the two processes. Typing messages with other types is only used on subterms of output messages, to ensure that the payload of an encryption or signature has the correct type, or to infer refinements so that we are able to determine the outcome of equality checks and destructor applications statically.

We present the typing rules for messages in Figure 2.4 and comment on them in the following.

Rules for nonces Confidential nonces can be given their label from the typing environment using rule TNonce. Since their label prevents them from being released in clear, the attacker cannot observe them and we do not need to add constraints for them. They can however be output in encrypted form and will then appear in the constraints associated to the encryption. Public nonces (labelled as LL) on the other hand are potentially already known by the attacker. Thus they can only be typed if they are equal on both sides (rule TNonceL), so as not to let the attacker observe a difference between each side.

Rules for variables and pairs We require variables to be the same in the two processes, deriving their type from the environment (TVar). The rule for pairs operates recursively component-wise (TPair).

$$\frac{\Gamma(n) = \tau_n^{l,a} \qquad \Gamma(m) = \tau_m^{l,a} \qquad l \in \{\texttt{HH}, \texttt{HL}\}}{\Gamma \vdash n \sim m : l \to \emptyset}\ (\textsc{TNonce})$$

$$\frac{\Gamma(n) = \tau_n^{\texttt{LL},a}}{\Gamma \vdash n \sim n : \texttt{LL} \to \emptyset}\ (\textsc{TNonceL}) \quad \frac{a \in \mathcal{C} \cup \mathcal{FN} \cup \mathcal{FK}}{\Gamma \vdash a \sim a : \texttt{LL} \to \emptyset}\ (\textsc{TCstFN})$$

$$\frac{\Gamma \vdash M \sim N : T \to \emptyset \qquad \exists T', l.T <: \text{key}^l(T')}{\Gamma \vdash \texttt{pk}(M) \sim \texttt{pk}(N) : \text{pkey}(T) \to \emptyset}\ (\textsc{TPubKey}) \quad \frac{k \in \text{keys}(\Gamma) \cup \mathcal{FK}}{\Gamma \vdash \texttt{pk}(k) \sim \texttt{pk}(k) : \texttt{LL} \to \emptyset}\ (\textsc{TPubKeyL})$$

$$\frac{\Gamma \vdash M \sim N : T \to \emptyset \qquad \exists T', l.T <: \text{key}^l(T')}{\Gamma \vdash \texttt{vk}(M) \sim \texttt{vk}(N) : \text{vkey}(T) \to \emptyset}\ (\textsc{TVKey}) \quad \frac{k \in \text{keys}(\Gamma) \cup \mathcal{FK}}{\Gamma \vdash \texttt{vk}(k) \sim \texttt{vk}(k) : \texttt{LL} \to \emptyset}\ (\textsc{TVKeyL})$$

$$\frac{\Gamma(k, k') = T}{\Gamma \vdash k \sim k' : T \to \emptyset}\ (\textsc{TKey}) \quad \frac{\Gamma(x) = T}{\Gamma \vdash x \sim x : T \to \emptyset}\ (\textsc{TVar})$$

$$\frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M' \sim N' : T' \to c'}{\Gamma \vdash \langle M, M' \rangle \sim \langle N, N' \rangle : T * T' \to c \cup c'}\ (\textsc{TPair})$$

$$\frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M' \sim N' : T' \to c' \qquad T' = \texttt{LL} \ \vee\ (\exists T'', l.T' <: \text{key}^l(T''))}{\Gamma \vdash \texttt{enc}(M, M') \sim \texttt{enc}(N, N') : (T)_{T'} \to c \cup c'}\ (\textsc{TEnc})$$

$$\frac{\Gamma \vdash M \sim N : (T)_{T'} \to c \qquad T' <: \text{key}^{\texttt{HH}}(T)}{\Gamma \vdash M \sim N : \texttt{LL} \to c \cup \{M \sim N\}}\ (\textsc{TEncH}) \quad \frac{\Gamma \vdash M \sim N : (\texttt{LL})_T \to c \qquad T <: \text{key}^{\texttt{LL}}(T') \text{ or } T = \texttt{LL}}{\Gamma \vdash M \sim N : \texttt{LL} \to c}\ (\textsc{TEncL})$$

$$\frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M' \sim N' : T' \to c' \qquad T' = \texttt{LL} \ \vee\ (\exists T'', T''', l.T' = \text{pkey}(T'') \wedge T'' <: \text{key}^l(T'''))}{\Gamma \vdash \texttt{aenc}(M, M') \sim \texttt{aenc}(N, N') : \{T\}_{T'} \to c \cup c'}\ (\textsc{TAenc})$$

$$\frac{\Gamma \vdash M \sim N : \{T\}_{\text{pkey}(T')} \to c \qquad T' <: \text{key}^{\texttt{HH}}(T)}{\Gamma \vdash M \sim N : \texttt{LL} \to c \cup \{M \sim N\}}\ (\textsc{TAencH})$$

$$\frac{\Gamma \vdash M \sim N : \{\texttt{LL}\}_T \to c \qquad (T = \text{pkey}(T') \wedge T' <: \text{eqkey}^l(T'')) \text{ or } T = \texttt{LL}}{\Gamma \vdash M \sim N : \texttt{LL} \to c}\ (\textsc{TAencL})$$

$$\frac{\Gamma \vdash M \sim N : T \to c \qquad \Gamma \vdash M \sim N : \texttt{LL} \to c' \qquad \Gamma \vdash M' \sim N' : \text{eqkey}^{\texttt{HH}}(T) \to \emptyset}{\Gamma \vdash \texttt{sign}(M, M') \sim \texttt{sign}(N, N') : \texttt{LL} \to c \cup c' \cup \{\texttt{sign}(M, M') \sim \texttt{sign}(N, N')\}}\ (\textsc{TSignH})$$

$$\frac{\Gamma \vdash M \sim N : \texttt{LL} \to c \qquad \Gamma \vdash M' \sim N' : \texttt{LL} \to c'}{\Gamma \vdash \texttt{sign}(M, M') \sim \texttt{sign}(N, N') : \texttt{LL} \to c \cup c'}\ (\textsc{TSignL})$$

$$\frac{\Gamma \vdash M \sim N : \texttt{HL} \to \emptyset}{\Gamma \vdash \texttt{h}(M) \sim \texttt{h}(N) : \texttt{LL} \to \{\texttt{h}(M) \sim \texttt{h}(N)\}}\ (\textsc{THash}) \quad \frac{\Gamma \vdash M \sim N : \texttt{LL} \to c}{\Gamma \vdash \texttt{h}(M) \sim \texttt{h}(N) : \texttt{LL} \to c}\ (\textsc{THashL})$$

$$\frac{\begin{array}{c} M{\downarrow} \neq \bot \qquad N{\downarrow} \neq \bot \\ \text{names}(M) \cup \text{names}(N) \cup \text{vars}(M) \cup \text{vars}(N) \cup \text{keys}(M) \cup \text{keys}(N) \subseteq \\ \text{dom}(\Gamma) \cup \text{keys}(\Gamma) \cup \mathcal{FN} \cup \mathcal{FK} \end{array}}{\Gamma \vdash M \sim N : \texttt{HL} \to \emptyset}\ (\textsc{THigh})$$

$$\frac{\Gamma \vdash M \sim N : T' \to c \qquad T' <: T}{\Gamma \vdash M \sim N : T \to c}\ (\textsc{TSub}) \quad \frac{\Gamma \vdash M \sim N : T \to c}{\Gamma \vdash M \sim N : T \vee T' \to c}\ (\textsc{TOr})$$

$$\frac{\begin{array}{l} \Gamma(m) = \tau_m^{l,1} \quad \text{or} \quad m \in \mathcal{FN} \cup \mathcal{C} \ \wedge\ l = \texttt{LL} \\ \Gamma(n) = \tau_n^{l',1} \quad \text{or} \quad n \in \mathcal{FN} \cup \mathcal{C} \ \wedge\ l' = \texttt{LL} \end{array}}{\Gamma \vdash m \sim n : [\![\tau_m^{l,1}\,;\, \tau_n^{l',1}]\!] \to \emptyset}\ (\textsc{TLR}^1) \quad \frac{\Gamma(m) = \tau_m^{l,\infty} \qquad \Gamma(n) = \tau_n^{l',\infty}}{\Gamma \vdash m \sim n : [\![\tau_m^{l,\infty}\,;\, \tau_n^{l',\infty}]\!] \to \emptyset}\ (\textsc{TLR}^\infty)$$

$$\frac{\Gamma \vdash M \sim N : [\![\tau_m^{l,a}\,;\, \tau_n^{l,a}]\!] \to c \qquad l \in \{\texttt{HL}, \texttt{HH}\}}{\Gamma \vdash M \sim N : l \to c}\ (\textsc{TLR'}) \quad \frac{\Gamma \vdash M \sim N : [\![\tau_n^{\texttt{LL},a}\,;\, \tau_n^{\texttt{LL},a}]\!] \to c}{\Gamma \vdash M \sim N : \texttt{LL} \to c}\ (\textsc{TLRL'})$$

$$\frac{\Gamma \vdash x \sim x : [\![\tau_m^{l,1}\,;\, \tau_n^{l',1}]\!] \to \emptyset \qquad \Gamma \vdash y \sim y : [\![\tau_{m'}^{l'',1}\,;\, \tau_{n'}^{l''',1}]\!] \to \emptyset}{\Gamma \vdash x \sim y : [\![\tau_m^{l,1}\,;\, \tau_{n'}^{l''',1}]\!] \to \emptyset}\ (\textsc{TLRVar})$$

Figure 2.4 – Rules for Messages

Rules for keys A non-standard rule that is crucial for the typing of branching protocols is rule TKey. As the typing environment contains types for bikeys (k, k'), this rule allows us to type two potentially different keys with their type from the environment.

With rules TPubKeyL and TVKeyL, we can give type $\mathtt{LL}$ to a public key or verification key that is explicitly associated with the same key on both sides. Rules TPubKey and TVKey on the other hand allow us to type different public keys $\mathtt{pk}(M), \mathtt{pk}(N)$ (resp. verification keys) with a public key type. They require that we can show there exists a valid key type for the terms M and N. This highlights another important technical point: we do not only support a fixed set of keys, but also allow for the usage of keys in variables, that have been received from the network. That is, the messages M, N do not need to explicitly be written as keys in the process, but rather might be variables containing for instance messages obtained from the network, or by decrypting messages received from the network.

Rules for symmetric encryption To show that a message is of type $(T)_{T'}$ – that is, a message of type T encrypted symmetrically with a key of type T', we have to show that the corresponding terms (used as plaintext and key) have exactly these types, in rule TEnc. The generated constraints are simply propagated. In addition we need to show that T' is either (a subtype of) a valid type for a key, or $\mathtt{LL}$. The first case corresponds to encryption performed by the processes using a message which can statically be shown to be a key. The second case models encryption with untrusted keys received from the network. Note that this rule allows us to encrypt messages with different keys in the two processes. It can for instance be used to type a message encrypted with k on the left and a different key k' on the right, provided the bikey (k, k') can be given a key type. This can be done *e.g.* if the typing environment Γ contains a mapping for (k, k'), using rule TKey.

For encryptions with honest keys, *i.e.* keys that can be given a key type with label $\mathtt{HH}$, we can use rule TEncH to give type $\mathtt{LL}$ to the messages. This rule requires that we can prove the payload type is respected. More precisely, once two messages have been given type $(T)_{T'}$, *e.g.* using rule TEnc, if T' is (a subtype of) $\mathrm{key}^{\mathtt{HH}}(T)$, then the messages can be given type $\mathtt{LL}$. Indeed, the label $\mathtt{HH}$ guarantees that the key is secret, and therefore hides the plaintexts from the attacker: thus we do not need to give them type $\mathtt{LL}$. However, the type T of the payload in $(T)_{T'}$ must be the same as the type T of the messages the key can encrypt in $\mathrm{key}^{\mathtt{HH}}(T)$. This enforces that keys always encrypt messages of the correct type specified in their key types. As we will see in the next section, this property will be used to extract information regarding the type of the plaintext when decrypting messages. In addition, even though the attacker cannot read the plaintext, he can perform an equality check on the ciphertext that he observes. Therefore, we add the entire ciphertexts to the constraints

Rule TEncL allows us to give type $\mathtt{LL}$ to encryptions even if the key is corrupted. However, we then have to type the plaintexts with type $\mathtt{LL}$ since we cannot guarantee their confidentiality. This typically models the case of messages encrypted with keys known by the attacker in sessions with dishonest users. Formally, this typing rule lets us give type $\mathtt{LL}$ to messages that already have type $(T)_{T'}$, in the case where $T' <: \mathrm{key}^{\mathtt{LL}}(T'')$ (for some T') is a subtype of a non-secret key type; or if $T' = \mathtt{LL}$ is not even a key type. It requires that the payload type T is $\mathtt{LL}$. Since we already add constraints for giving type $\mathtt{LL}$ to the plaintext, we do not need to add any additional constraints regarding the ciphertexts.

Rules for asymmetric encryption The case of asymmetric encryption is quite similar, although slightly more involved. The difference is that we can always choose to ignore the key

type and use type LL to check the payload instead. This allows us to type messages produced by the attacker, who, contrary to the symmetric case, always has access to the public key but does not need to respect its type, even for honest keys. He may use the public key to encrypt plaintexts whose type do not match the one specified in the key type. However, any messages encrypted this way are already known by the attacker, and must therefore be of type LL. Therefore, we allow giving type LL to encryptions where the payload does not have the expected type even when the key is honest, provided the payload itself can be given type LL.

As in the symmetric case, to show that a message is of type $\{T\}_{T'}$, *i.e.* the type for messages of type T encrypted asymmetrically with a key of type T', rule TAENC requires that the plaintext and key can be given type T and T' respectively. In addition T' must be is either a valid type for a public key, or LL. Here again, the second case models encryption with untrusted keys received from the network.

When encrypting with honest keys (label HH), rule TAENCH, similarly to rule TENCH, lets us give type LL to messages of type $\{T\}_{T'}$, provided that $\mathcal{T}'$ is a public key type associated with a confidential key, and that T is indeed the payload type specified in the key type. Here also we add the entire encryptions to the constraints, since the attacker can only check different encryptions for equality, but not open them to reveal the plaintext.

Rule TAENCL is similar to rule TENCL, except that, as explained, the encryption key is public, even when the associated decryption key is secret, *i.e.* if the key has type $\mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(T))$. The attacker may use it to encrypt messages of the "wrong" type, that is, a type different from T. Hence rule TAENCL allows us to give type LL to encryptions even if we do not respect the payload type, or if the key is corrupted. We then have to type the plaintexts with type LL since we cannot guarantee their confidentiality (when the encryption is produced by the protocol agents) or since they are already known by the attacker (when it is produced by him). Additionally, we have to ensure that the same key is used in both processes, because the attacker might possess the corresponding private keys and test which decryption succeeds. This is done by requiring that the private key has type $\mathrm{eqkey}^{l}(T)$ (rather than just $\mathrm{key}^{l}(T)$). As in the symmetric case, since we already add constraints for giving type LL to the plaintext, we do not need to add any additional constraints.

Rules for signatures Signatures are also handled similarly, with the difference that, contrary to encryption, they do not hide the contents of the message. Therefore we need not only to ensure that the message to be signed correctly has the type specified by the key type, but also to give it type LL even if an honest key is used, as outputting the signature would reveal the payload to the attacker.

Formally, rule TSIGNH lets us type signatures with honest keys. We can give type LL to messages $\mathtt{sign}(M, M')$ on the left and $\mathtt{sign}(N, N')$ on the right, provided that

- the terms used as signing keys (M', N') can be given a key type $\mathrm{eqkey}^{\mathtt{HH}}(T)$ (for some T);
- the payloads M, N can be given the type T, which the key type specifies as the expected payload type for these keys;
- the payloads can also be given type LL, as they could be seen by the attacker.

Note that the key type is required to be $\mathrm{eqkey}^{\mathtt{HH}}(T)$ and not just $\mathrm{key}^{\mathtt{HH}}(T)$: indeed, the attacker has access to all verification keys, and thus would notice messages being signed with different keys on the left and on the right. We additionally collect together all the constraints generated when typing the payloads.

Rule TSignL is similar for the case of dishonest keys, that have type LL, except that in that case the payload type is not enforced, as the attacker may sign messages of any type.

Rules for hashes The case of hashing can be seen as similar to asymmetric encryption with honest keys, but is simpler, as we do not have to take key types into account. The first typing rule for hashes (THash) gives them type LL and adds the term to the constraints, only requiring that the arguments of the hash function can be given type HL, the least restrictive type. Intuitively this is justified, because the hash function makes it impossible to recover the argument. Thus, publishing a hash of any messages is fine regardless of their types: the hash simply must be added to the constraint, as the attacker may still perform equality checks on hashes he sees.

The second rule (THashL) gives type LL only if we can also give type LL to the argument of the hash function, but does not add any constraints on its own, and just passes on the constraints created for the arguments. This means we are typing the message as if the hash function would not have been applied and use the message without the hash, which is a strictly stronger result. Both rules have their applications: while the former has to be used whenever we hash a secret, the latter may be useful to avoid the creation of unnecessary constraints when hashing terms like constants or public nonces.

Rules for HL, subtyping, and union types Rule THigh states that we can give type HL to any message, which intuitively means that it is always safe to treat any message as if it should not be published. The rule simply enforces that the message is well-formed in the sense that it only contains names, variables, and keys that are either bound in the typing environment, or free (*i.e.* generated by the attacker).

As expected, rule TSub allows us to subtype messages according to the subtyping relation.

Rule TOr also follows the intuition: it allows us to give a union type to messages, as long as they are typable with at least one of the two types.

Rules for refinements The remaining rules deal with refinement types. TLR^1 and TLR^∞ are the introduction rules for them. Two names m, n can be given type $[\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!]$ if $\tau_m^{l,a}$ and $\tau_n^{l',a}$ are their respective type in Γ. The same is true for constant and public names: they do not have a type in Γ and are instead given label LL. Nonce generated by the protocol can be replicated or not (both rules TLR^1 and TLR^∞ apply), while public names and constant cannot be marked as replicated (only rule TLR^1 applies to them).

Rules TLR' and TLRL' are the corresponding elimination rules: similarly to rules TNonce and TNonceL, they allow to give type $l \in \{\texttt{HH}, \texttt{HL}, \texttt{LL}\}$ to messages with a refinement type with the corresponding label $[\![\tau_m^{l,a}\,;\,\tau_n^{l,a}]\!]$. When $l = \texttt{LL}$, they additionally require that the two names m and n are the same, just as in TNonceL.

Finally, TLRVar allows to derive a new refinement type for two variables for which we have singleton refinement types, by taking the left refinement of the left variable and the right refinement of the right variable. Formally, if x has type $[\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!]$ and y has type $[\![\tau_{m'}^{l'',1}\,;\,\tau_{n'}^{l''',1}]\!]$, then a message consisting of x on the left and y on the right can be given type $[\![\tau_m^{l,1}\,;\,\tau_{n'}^{l''',1}]\!]$. The reason why this rule only applies with non-replicated refinement types lies in the intuition of the replicated types. A replicated type $[\![\tau_m^{l,\infty}\,;\,\tau_n^{l',\infty}]\!]$ is intended to represent the type for a nonce generated when invoking the randomness generator in one copy of a replicated process. More precisely, the index of the replicated copy in question is intended to be the same on both sides.

That is, this nonce should have been generated in the first replication on both sides, or in the second replication on both sides, etc. but not *e.g.* in the first on the left and the second on the right. Applying rule TLRVar with replicated refinement types would break this intuition: x and y could very well contain nonces generated in different replications of the process. Hence, we restrict it to unreplicated types.

We will see applications of this rule in the e-voting protocol, where we use it to combine A's vote (0 on the left, 1 on the right) and B's vote (1 on the left, 0 on the right), into a message that is the same on both sides.

2.3.4 Typing Processes

From now on, we assume that processes assign a type to freshly generated nonces and keys. That is, $\texttt{new}\ n.P$ is now of the form $\texttt{new}\ n : T.\ P$, and similarly for $\texttt{new}\ k.P$. This requires a (very light) type annotation from the user. In addition, we assume in this section that P and Q do not contain replication and that variables and names are renamed to avoid any capture.

The typing judgement for processes is of the form $\Gamma \vdash P \sim Q \to C$ and can be interpreted as follows: if two processes P and Q can be typed in Γ and if the generated constraint set C is consistent, then P and Q are trace equivalent.

When typing processes, the typing environment Γ is passed down and extended from the root towards the leaves of the syntax tree of the process, *i.e.*, following the execution semantics. The generated constraints C however, are passed up from the leaves towards the root, so that at the root we get all generated constraints, modelling the attacker's global view on the process execution.

More precisely, each possible execution path of the process - there may be multiple paths because of conditionals - creates its own set of constraints c together with the typing environment Γ that contains types for all names and variables appearing in c. Hence, typing a pair of processes P, Q yields a constraint set C, which, as defined in Section 2.3.2, is a set elements of the form (c, Γ) for a set of constraints c. Keeping track of the types assigned to variables when typing is required for the constraint checking procedure, as they help us to be more precise when checking the consistency of constraints.

Well-formed environment We first introduce a notion of *well-formedness* for typing environments. The judgement $\Gamma \vdash \diamond$ reads "Γ is well-formed", and is defined in Figure 2.5.

Basically, this condition ensures that the environment is correctly constructed. First, a well-formed environment Γ must only contain bindings for nonces, variables, and bikeys. It can also not contain several types for the same nonce, variable, or bikey. A name $n \in \mathcal{BN}$ can only be bound to a nonce type $\tau_n^{l,a}$ (rule GNonce); a variable x can be bound to any type (rule GVar); and a bikey $(k, k') \in \mathcal{BK}^2$ must be associated with a key type. The three rules for bikeys enforce the intuitive meaning of the different key types explained earlier (Section 2.1). They basically ensure that $\text{seskey}^{l,a}(T)$ and $\text{eqkey}^{l}(T)$ are only given when the key is the same on both sides, and that in addition a key with type $\text{seskey}^{l,a}(T)$ is never part of another bikey.

More precisely a bikey (k, k') can be bound to either

- a key type $\text{seskey}^{l,a}(T)$ for some a, l, T (rule GSesKey): in that case, k must be equal to k', and this key must not appear in any other bikey bound in Γ.
- a key type $\text{eqkey}^{l}(T)$ (rule GEqKey): as in the previous case, we require that $k = k'$. k may appear in other bikeys bound in Γ, but we require that any such bikey is given a key

$$[] \vdash \diamond \ (\text{GNil}) \qquad \frac{\Gamma \vdash \diamond \qquad n \in \mathcal{BN} \qquad n \notin \text{dom}(\Gamma)}{\Gamma, n : \tau_n^{l,a} \vdash \diamond} \ (\text{GNonce})$$

$$\frac{\Gamma \vdash \diamond \qquad x \notin \text{dom}(\Gamma)}{\Gamma, x : T \vdash \diamond} \ (\text{GVar})$$

$$\frac{\begin{array}{c}\Gamma \vdash \diamond \qquad k, k' \in \mathcal{BK} \qquad (k, k') \notin \text{dom}(\Gamma) \qquad k \notin \text{seskeys}(\Gamma) \qquad k' \notin \text{seskeys}(\Gamma) \\ (\forall l, T'.\ \Gamma(k,k) = \text{eqkey}^l(T') \ \Rightarrow \ l = \texttt{HH}) \qquad (\forall l, T'.\ \Gamma(k',k') = \text{eqkey}^l(T') \ \Rightarrow \ l = \texttt{HH})\end{array}}{\Gamma, (k, k') : \text{key}^{\texttt{HH}}(T) \vdash \diamond} \ (\text{GKey})$$

$$\frac{\begin{array}{c}\Gamma \vdash \diamond \qquad k \in \mathcal{BK} \qquad (k, k) \notin \text{dom}(\Gamma) \\ (\forall k', l', T'.\ \Gamma(k,k') <: \text{key}^l(T') \ \Rightarrow \ l = l') \qquad (\forall k', l', T'.\ \Gamma(k',k) <: \text{key}^l(T') \ \Rightarrow \ l = l')\end{array}}{\Gamma, (k, k) : \text{eqkey}^l(T) \vdash \diamond} \ (\text{GEqKey})$$

$$\frac{\Gamma \vdash \diamond \qquad k \in \mathcal{BK} \qquad \forall k' \neq k.(k, k') \notin \text{dom}(\Gamma) \ \wedge \ (k', k) \notin \text{dom}(\Gamma)}{\Gamma, (k, k) : \text{seskey}^{l,a}(T) \vdash \diamond} \ (\text{GSesKey})$$

Figure 2.5 – Well-formedness of the typing environment

type with the same security label l. It would indeed not make sense for a key to be secret when considered in one bikey, and known by the attacker in another bikey. Note that the condition that (k, k) itself is not already bound in Γ implies in particular that k does not already have type $\text{seskey}^{l,a}(T)$.

- a key type $\text{key}^l(T)$ (rule GKey). In that case, we allow k and k' to be different, but we require that $l = \texttt{HH}$: indeed, a bikey with a different value on either side must be kept secret, or the attacker could use it to distinguish the two processes. As before, we require that neither k nor k' already appear with a $\text{seskey}^{l,a}(T)$ type, and that any other binding for k or k' has the same label HH (as rule GKey already implies this last condition for type $\text{key}^l(T)$, we only need to check it when the type is $\text{eqkey}^l(T)$).

Typing processes: inputs, outputs, parallel branching Our typing rules for processes are presented in Figure 2.6 and explained in the following. Rule PZero copies the current typing environment in the constraints and checks the well-formedness of the environment ($\Gamma \vdash \diamond$). As this instruction is at the end of every execution path, we use the typing environment at this point for constraint checking, as it is guaranteed to contain an entry for all nonces and variables that may have been used on this execution path and hence may appear in the constraints. In addition, we require that all variables with union types in Γ have been split beforehand, *i.e.* that $\text{branches}(\Gamma) = \{\Gamma\}$. As we will see, this can be done by applying rule POr as needed when reaching the null process.

Messages output on the network are possibly learned by the attacker. Rule POut states that we can output messages to the network if we can type them with type LL, *i.e.*, they are indistinguishable to the attacker, provided that the generated set c of constraints is consistent. The constraints of c are then added to all constraints in the constraint set C, using the $\cup_\forall$ operator defined earlier in Section 2.3.2.

$$\frac{\Gamma \vdash \diamond \qquad \text{branches}(\Gamma) = \{\Gamma\}}{\Gamma \vdash 0 \sim 0 \to (\emptyset, \Gamma)}\ (\textsc{PZero})$$

$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c}{\Gamma \vdash \mathtt{out}(M).P \sim \mathtt{out}(N).Q \to C \cup_\forall c}\ (\textsc{POut}) \qquad \frac{\Gamma, x : \mathtt{LL} \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{in}(x).P \sim \mathtt{in}(x).Q \to C}\ (\textsc{PIn})$$

$$\frac{\Gamma, n : \tau_n^{l,a} \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{new}\ n : \tau_n^{l,a}.P \sim \mathtt{new}\ n : \tau_n^{l,a}.Q \to C}\ (\textsc{PNew})$$

$$\frac{\Gamma, (k,k) : \text{seskey}^{l,a}(T) \vdash P \sim Q \to C}{\Gamma \vdash \mathtt{new}\ k : \text{seskey}^{l,a}(T).P \sim \mathtt{new}\ k : \text{seskey}^{l,a}(T).Q \to C}\ (\textsc{PNewKey})$$

$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash P \mid P' \sim Q \mid Q' \to C \cup_\times C'}\ (\textsc{PPar}) \qquad \frac{\Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma, x : T' \vdash P \sim Q \to C'}{\Gamma, x : T \vee T' \vdash P \sim Q \to C \cup C'}\ (\textsc{POr})$$

$$\frac{\Gamma \vdash t \sim t' : T \qquad \Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = t'\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to C \cup C'}\ (\textsc{PLet})$$

$$\frac{\begin{array}{c}\Gamma(y) = \mathtt{LL} \qquad \Gamma(k_1,k_2) <: \text{key}^{\mathtt{HH}}(T) \\ \Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C' \\ (\forall T'.\forall k_3 \neq k_2.\ \Gamma(k_1,k_3) <: \text{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P \sim Q' \to C_{k_3}) \\ (\forall T'.\forall k_3 \neq k_1.\ \Gamma(k_3,k_2) <: \text{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P' \sim Q \to C'_{k_3})\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{let}\ x = \mathtt{dec}(y,k_1)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = \mathtt{dec}(y,k_2)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to \\ \qquad C \cup C' \cup (\bigcup_{k_3} C_{k_3}) \cup (\bigcup_{k_3} {C'}_{k_3})\end{array}}\ (\textsc{PLetDec})$$

$$\frac{\begin{array}{c}\Gamma(y) = \mathtt{LL} \qquad \Gamma(k,k) <: \text{key}^{\mathtt{HH}}(T) \qquad \Gamma, x : T \vdash P \sim Q \to C \\ \Gamma, x : \mathtt{LL} \vdash P \sim Q \to C' \qquad \Gamma \vdash P' \sim Q' \to C'' \\ (\forall T'.\forall k_3 \neq k.\ \Gamma(k,k_3) <: \text{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P \sim Q' \to C_{k_3}) \\ (\forall T'.\forall k_3 \neq k.\ \Gamma(k_3,k) <: \text{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P' \sim Q \to C'_{k_3})\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{let}\ x = \mathtt{adec}(y,k)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = \mathtt{adec}(y,k)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to \\ \qquad C \cup C' \cup C'' \cup (\bigcup_{k_3} C_{k_3}) \cup (\bigcup_{k_3} {C'}_{k_3})\end{array}}\ (\textsc{PLetAdecSame})$$

$$\frac{\begin{array}{c}k_1 \neq k_2 \qquad \Gamma(y) = \mathtt{LL} \qquad \Gamma(k_1,k_2) <: \text{key}^{\mathtt{HH}}(T) \\ \Gamma, x : T \vdash P \sim Q \to C \qquad \Gamma, x : \mathtt{LL} \vdash P \sim Q' \to C' \\ \Gamma, x : \mathtt{LL} \vdash P' \sim Q \to C'' \qquad \Gamma \vdash P' \sim Q' \to C''' \\ (\forall T'.\forall k_3 \neq k_2.\ \Gamma(k_1,k_3) <: \text{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P \sim Q' \to C_{k_3}) \\ (\forall T'.\forall k_3 \neq k_1.\ \Gamma(k_3,k_2) <: \text{key}^{\mathtt{HH}}(T') \Rightarrow \Gamma, x : T' \vdash P' \sim Q \to C'_{k_3})\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{let}\ x = \mathtt{adec}(y,k_1)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = \mathtt{adec}(y,k_2)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to \\ \qquad C \cup C' \cup C'' \cup C''' \cup (\bigcup_{k_3} C_{k_3}) \cup (\bigcup_{k_3} {C'}_{k_3})\end{array}}\ (\textsc{PLetAdecDiff})$$

$$\frac{\Gamma(y) = [\![\tau_n^{l,a}\ ;\ \tau_m^{l',a}]\!] \vee \Gamma(y) <: \text{key}^{l}(T) \qquad \Gamma \vdash P' \sim Q' \to C'}{\Gamma \vdash \mathtt{let}\ x = d(y)\ \mathtt{in}\ P\ \mathtt{else}\ P' \sim \mathtt{let}\ x = d(y)\ \mathtt{in}\ Q\ \mathtt{else}\ Q' \to C'}\ (\textsc{PLetLR})$$

Figure 2.6 – Rules for processes (1)

$$\frac{\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C' \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c \qquad \Gamma \vdash M' \sim N' : \mathtt{LL} \to c'}{\Gamma \vdash \mathtt{if}\ M = M'\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = N'\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to (C \cup C') \cup_\forall (c \cup c')}\ (\textsc{PIfL})$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!] \to \emptyset \qquad \Gamma \vdash M_2 \sim N_2 : [\![\tau_{m'}^{l'',1}\,;\,\tau_{n'}^{l''',1}]\!] \to \emptyset \\ b = (\tau_m^{l,1} \stackrel{?}{=} \tau_{m'}^{l'',1}) \qquad b' = (\tau_n^{l',1} \stackrel{?}{=} \tau_{n'}^{l''',1}) \qquad \Gamma \vdash P_b \sim Q_{b'} \to C\end{array}}{\Gamma \vdash \mathtt{if}\ M_1 = M_2\ \mathtt{then}\ P_\top\ \mathtt{else}\ P_\bot \sim \mathtt{if}\ N_1 = N_2\ \mathtt{then}\ Q_\top\ \mathtt{else}\ Q_\bot \to C}\ (\textsc{PIfLR})$$

$$\frac{\Gamma \vdash P' \sim Q' \to C' \qquad \Gamma \vdash M \sim N : \mathtt{LL} \to c \qquad \Gamma \vdash M' \sim N' : \mathtt{HH} \to c'}{\Gamma \vdash \mathtt{if}\ M = M'\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = N'\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to C'}\ (\textsc{PIfS})$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,\infty}\,;\,\tau_n^{l',\infty}]\!] \to \emptyset \qquad \Gamma \vdash M_2 \sim N_2 : [\![\tau_m^{l,\infty}\,;\,\tau_n^{l',\infty}]\!] \to \emptyset \\ \Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C'\end{array}}{\Gamma \vdash \mathtt{if}\ M_1 = M_2\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N_1 = N_2\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to C \cup C'}\ (\textsc{PIfLR*})$$

$$\frac{\begin{array}{c}\Gamma \vdash P \sim Q \to C \qquad \Gamma \vdash P' \sim Q' \to C' \\ \Gamma \vdash M \sim N : \mathtt{LL} \to c \qquad \Gamma \vdash t \sim t : \mathtt{LL} \to c' \qquad t \in \mathcal{K} \cup \mathcal{N} \cup \mathcal{C}\end{array}}{\Gamma \vdash \mathtt{if}\ M = t\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = t\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to C \cup C'}\ (\textsc{PIfP})$$

$$\frac{\Gamma \vdash P' \sim Q' \to C' \qquad \Gamma \vdash M \sim N : T * T' \to c \qquad \Gamma \vdash M' \sim N' : [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!] \to \emptyset}{\Gamma \vdash \mathtt{if}\ M = M'\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = N'\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to C'}\ (\textsc{PIfI})$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!] \to \emptyset \qquad \Gamma \vdash M_2 \sim N_2 : [\![\tau_{m'}^{l'',a'}\,;\,\tau_{n'}^{l''',a'}]\!] \to \emptyset \\ \tau_m^{l,a} \neq \tau_{m'}^{l'',a'}, \tau_n^{l',a} \neq \tau_{n'}^{l''',a'} \qquad \Gamma \vdash P' \sim Q' \to C\end{array}}{\Gamma \vdash \mathtt{if}\ M_1 = M_2\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N_1 = N_2\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to C}\ (\textsc{PIfLR'*})$$

$$\frac{\Gamma \vdash P \sim Q \to C_1 \qquad \Gamma \vdash P \sim Q' \to C_2 \qquad \Gamma \vdash P' \sim Q \to C_3 \qquad \Gamma \vdash P' \sim Q' \to C_4}{\Gamma \vdash \mathtt{if}\ M = M'\ \mathtt{then}\ P\ \mathtt{else}\ P' \sim \mathtt{if}\ N = N'\ \mathtt{then}\ Q\ \mathtt{else}\ Q' \to C_1 \cup C_2 \cup C_3 \cup C_4}\ (\textsc{PIfAll})$$

Figure 2.7 – Rules for processes (2)

$$\frac{\Gamma(k,k) <: \mathrm{key}^{\mathtt{LL}}(T) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{dec}(x,k) \sim \mathtt{dec}(x,k) : \mathtt{LL}} \text{ (DDecL)}$$

$$\frac{\Gamma(k,k) = \mathrm{seskey}^{l,a}(T') \qquad \Gamma(x) = (T)_{\mathrm{seskey}^{l,a}(T')}}{\Gamma \vdash \mathtt{dec}(x,k) \sim \mathtt{dec}(x,k) : T} \text{ (DDecT)}$$

$$\frac{\Gamma(y) = \mathrm{seskey}^{l,a}(T') \qquad \Gamma(x) = (T)_{\mathrm{seskey}^{l,a}(T')}}{\Gamma \vdash \mathtt{dec}(x,y) \sim \mathtt{dec}(x,y) : T} \text{ (DDecT')}$$

$$\frac{\Gamma(y) = \mathrm{seskey}^{\mathtt{HH},a}(T) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{dec}(x,y) \sim \mathtt{dec}(x,y) : T} \text{ (DDecH')}$$

$$\frac{(\Gamma(y) = \mathrm{seskey}^{\mathtt{LL},a}(T) \ \vee \ \Gamma(y) = \mathtt{LL}) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{dec}(x,y) \sim \mathtt{dec}(x,y) : \mathtt{LL}} \text{ (DDecL')}$$

$$\frac{\Gamma(k,k) <: \mathrm{key}^{\mathtt{LL}}(T) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{adec}(x,k) \sim \mathtt{adec}(x,k) : \mathtt{LL}} \text{ (DAdecL)}$$

$$\frac{\Gamma(y) = \mathrm{seskey}^{\mathtt{HH},a}(T) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{adec}(x,y) \sim \mathtt{adec}(x,y) : T \ \vee \ \mathtt{LL}} \text{ (DAdecH')}$$

$$\frac{(\Gamma(y) = \mathrm{seskey}^{\mathtt{LL},a}(T) \ \vee \ \Gamma(y) = \mathtt{LL}) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{adec}(x,y) \sim \mathtt{adec}(x,y) : \mathtt{LL}} \text{ (DAdecL')}$$

$$\frac{\Gamma(k,k) = \mathrm{seskey}^{l,a}(T') \qquad \Gamma(x) = \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T'))}}{\Gamma \vdash \mathtt{adec}(x,k) \sim \mathtt{adec}(x,k) : T} \text{ (DAdecT)}$$

$$\frac{\Gamma(y) = \mathrm{seskey}^{l,a}(T') \qquad \Gamma(x) = \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T'))}}{\Gamma \vdash \mathtt{adec}(x,y) \sim \mathtt{adec}(x,y) : T} \text{ (DAdecT')}$$

$$\frac{\Gamma(k,k) <: \mathrm{key}^{\mathtt{HH}}(T) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{checksign}(x, \mathtt{vk}(k)) \sim \mathtt{checksign}(x, \mathtt{vk}(k)) : T} \text{ (DCheckH)}$$

$$\frac{\Gamma(k,k) <: \mathrm{key}^{\mathtt{LL}}(T) \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{checksign}(x, \mathtt{vk}(k)) \sim \mathtt{checksign}(x, \mathtt{vk}(k)) : \mathtt{LL}} \text{ (DCheckL)}$$

$$\frac{\Gamma(y) = \mathrm{vkey}(T) \qquad T <: \mathrm{eqkey}^{\mathtt{HH}}(T') \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{checksign}(x,y) \sim \mathtt{checksign}(x,y) : T'} \text{ (DCheckH')}$$

$$\frac{(\Gamma(y) = \mathrm{vkey}(T) \wedge T <: \mathrm{eqkey}^{\mathtt{LL}}(T')) \text{ or } \Gamma(y) = \mathtt{LL} \qquad \Gamma(x) = \mathtt{LL}}{\Gamma \vdash \mathtt{checksign}(x,y) \sim \mathtt{checksign}(x,y) : \mathtt{LL}} \text{ (DCheckL')}$$

$$\frac{\Gamma(x) = T * T'}{\Gamma \vdash \pi_1(x) \sim \pi_1(x) : T} \text{ (DFst)} \quad \frac{\Gamma(x) = T * T'}{\Gamma \vdash \pi_2(x) \sim \pi_2(x) : T'} \text{ (DSnd)}$$

$$\frac{\Gamma(x) = \mathtt{LL}}{\Gamma \vdash \pi_1(x) \sim \pi_1(x) : \mathtt{LL}} \text{ (DFstL)} \quad \frac{\Gamma(x) = \mathtt{LL}}{\Gamma \vdash \pi_2(x) \sim \pi_2(x) : \mathtt{LL}} \text{ (DSndL)}$$

Figure 2.8 – Destructor Rules

Conversely, messages input from the network are stored in variables to which we give type LL (rule PIn), since the message is known to or may even originate from the attacker.

Rule PNew introduces a new nonce, which may be used in the continuation processes. This nonce is added with its type to the environment. Similarly, rule PNewKey allows us to generate new session keys at runtime, which is useful to accurately model security protocols.

In rule PPar, when typechecking parallel composition $P \mid P' \sim Q \mid Q'$, we first typecheck the individual subprocesses $P \sim Q$, $P' \sim Q'$. We then take the product union $C \cup_\times C'$ (as defined in Section 2.3.2) of the generated constraint sets C, C' as the new constraint set.

Intuitively, the elements in C correspond to the constraints for each possible execution path in P and Q, and the ones in C' to the execution paths of P' and Q'. Hence for their parallel composition, we need to combine each possible execution of P (and Q) with each possible execution of P' (and Q'). The product union operation does exactly that. The combinations that are discarded due to incompatible environments correspond to impossible executions (*e.g.*, taking the left branch in P and the right branch in P' in two conditionals with the same guard).

POr is the elimination rule for union types, which requires the continuation process to be well-typed with both types.

Generic rules for destructor application Several rules handle the application of destructors to messages: a general rule, PLet, and four additional rules for specific cases, PLetDec, PLetAdecSame, PLetAdecDiff, and PLetLR.

First, rule PLet applies to the case where we can statically know that a destructor application succeeds or fails equally in the two processes. To ensure this property, for this rule, we only allow the same destructor to be applied to the same variable in both processes. As usual, we then typecheck the two then-branches together, as well as the else branches, and then take the union of the corresponding constraints. Rule PLet relies on additional rules for destructors. We present them in Figure 2.8.

- Rule DDecL simply states that decrypting a variable of type LL (typically input from the network) with an untrusted key (label LL) yields a result of type LL. Rule DDecL' is similar, but can be applied when decrypting using a variable as a key, as opposed to decrypting with a message that is explicitly a key. Rules DAdecL and DAdecL' are analogous to rules DDecL and DDecL' respectively, for the case of asymmetric decryption.

- Symmetric decryption with a trusted session key (with label HH) returns, in case of success, a value of the key's payload type (rule DDecH'). Note that this rule only applies when the key has type $\mathrm{seskey}^{\mathtt{HH},a}(T)$. Indeed, using a trusted key is not sufficient to know that decryption will succeed or fail equally on both sides: we need to know that, when encrypting, this key can never be used to encrypt a message on the left while a different key would be used on the right (and conversely). This is enforced by requiring that the key never appears with another key in a bikey in Γ. Type $\mathrm{seskey}^{\mathtt{HH},a}(T)$ exactly expresses this condition. To decrypt with a key that does appear in other bikeys, a more complex rule (PLetDec) is required, which we will present later. Rule DDecH' only applies to the case where the key is a variable. No additional "DDecH" rule is used for the case where the key is explicit, as rule PLetDec will apply to all decryptions with trusted, explicit symmetric keys.

- Rule DAdecH' is similar to rule DDecH', but for the asymmetric case. As stated earlier, in that case, even when decrypting with a trusted key, the ciphertext may have been

constructed by the attacker who knows the public key. Hence, decryption yields a message of type $T \vee \texttt{LL}$, rather than simply T. Here again it is important that the key is of type $\text{seskey}^{\texttt{HH},a}(T)$, since this guarantees that the key is never used in combination with a different key and hence decryption will always equally succeed or fail in both processes. As before, the other cases are handled by special rules PLetAdecSame and PLetAdecDiff.

- Rule DDecT treats the case in which we know that the variable x is a symmetric encryption of a specific type. If the type of the key used for decryption matches the key type used for encryption, we know the exact type of the result of a successful decryption. DDecT' is similar to DDecT, with a variable as key. Asymmetric decryption is handled by analogous rules DAdecT and DAdecT'.

- Verifying a signature with the verification key associated to a trusted key yields a message of the payload type specified in the type of the key in rule DCheckH. If the signature key is not trusted, we simply obtain an untrusted message, of type LL, as the signature may have been produced by the attacker (rule DCheckL). Note that the typing rules for messages only allow to sign messages with the same key on either side, as using a different key would let the attacker distinguish between the two sides. Hence, as long as the signature is checked using the same key on both sides, we know that the verification will succeed or fail equally on both sides.

  Rules DCheckH' and DCheckL' are similar, except a variable is used as a verification key.

- Finally, rules DFst, DSnd apply when projecting pairs: if a message has type $T * T'$, its first and second projections respectively have types T and T'. Rules DFstL and DSndL state that projecting an untrusted message of type LL yields another untrusted message.

Rules for symmetric decryption Rule PLetDec handles symmetric decryptions where we use fixed honest keys k_1, k_2 (labelled HH as a bikey in Γ) for decryption in each process. In our type system, we allow encryptions with potentially different keys on either side, which requires cross-case validation in order to retain soundness. That is, since different keys may be used, we cannot statically know that decryption will succeed or fail equally on both sides. For instance decrypting with k_1 on the left and k_2 on the right a message encrypted with k_1 on the left and some other key k_3 on the right would succeed on the left but fail on the right. Still, the number of possible combinations of encryption keys is limited by the assignments in the typing environment Γ. The typing rules for messages intuitively ensure that the messages we encounter can only be encrypted with keys that match one of the bikeys in Γ. To cover all the possibilities, we thus type the following combinations of continuation processes:

- Both `then` branches: In this case we know that decryption succeeds on both sides, *i.e.* key k_1 was used for encryption on the left, and k_2 on the right. Since $\Gamma(k_1, k_2) <: \text{key}^{\texttt{HH}}(T)$, we know that in this case the payload type is T, and we type the continuation with $\Gamma, x : T$.

- Both `else` branches: If decryption fails on both sides, we type the two `else` branches without introducing any new variables.

- Left `then`, right `else`: The encryption may have been created with key k_1 on the left side and another key than k_2 on the right side. Hence, we consider each $k_3 \neq k_2$ such that

$\Gamma(k_1, k_3)$ maps to a key type with some payload type T'. Note that by well-formedness of Γ, all possible such bikeys also have label `HH`. For each such k', we have to typecheck the left `then` branch and the right `else` branch with $\Gamma, x : T'$.

- Left `else`, right `then`: This case is analogous to the previous one.

The generated set of constraints is then simply the union of all generated constraints for the subprocesses, accounting for all possible execution paths.

Rules for asymmetric decryption The rules for asymmetric decryption are similar, except they are split into two rules, PLetAdecSame and PLetAdecDiff, depending on whether the decryption key is the same on both sides or not. As in the symmetric case, we cannot know that decryption succeeds or fails equally on the left and the right, and we thus need to consider all bikeys in Γ where the decryption keys occur. The difference lies in the fact that, contrary to the symmetric case, even when the key is trusted, the attacker knows the public key, and may thus have produced the ciphertext.

If the same decryption key is used on both sides, when decryption succeeds on both sides, it may yield a plaintext provided by the attacker. Hence we additionally typecheck both then-branches, giving type `LL` to the plaintext. However when decryption fails *e.g.* on the left and succeeds on the right, this means the ciphertext we tried to decrypt was not encrypted with the same public key on both sides. Thus it cannot originate from the attacker, who has to behave the same on both sides. We therefore do not need to typecheck the cross-cases with a plaintext of type `LL` (but only with the payload type specified in the key types).

If different keys k_1, k_2 are used to decrypt, when decryption succeeds on both sides, we know for the same reason that the message was not constructed by the attacker, and thus we do not need to typecheck the then-branches with type `LL`. However we might be decrypting a message from the attacker, encrypted with *e.g.* $\mathtt{pk}(k_1)$ on both sides. Then decryption would then succeed on the left and fail on the right: we need to typecheck the left then-branch together with the right else-branch, while considering the plaintext to be `LL` (and conversely for the case of messages encrypted with $\mathtt{pk}(k_2)$).

In the special case in which we know that the concrete value of the argument of the destructor application is a nonce or constant due to a refinement type, and we know statically that any destructor application will fail, we only need to type-check the else branch (rule PLetLR).

Rules for conditional branching The remaining rules handle conditional branching. As for destructor applications, the difficulty while typing conditionals is to determine whether the same branch is taken in both processes. One way to ensure this is with a trick: in rule PIfL, we type both the left and the right operands of the conditional with type `LL` and add both generated sets of constraints to the constraint set. Intuitively, this means that the attacker could perform the equality test himself, since the guard is of type `LL`, which means that the conditional must take the same branch on the left and on the right. These are exactly the same steps that we would take to output the respective values on the network (see rule POut), hence it would be safe to output them on the network. It then follows directly that the equality check evaluates to the same value in the two processes, otherwise the attacker could use his observations on the (theoretical) outputs to distinguish the processes.

Rule PIfP allows us to do this without generating new constraints, in the special case where one of the messages being compared is the same public name or constant on both sides. Indeed

in that case, regardless of constraints, the other message (of type LL) has to be either equal to that name on both sides, or different from it on both sides.

In the special case in which we can statically determine the concrete value of the terms in the conditional, thanks to the refinement types, we have to typecheck only the single combination of branches that will be executed (rule PIFLR). Note that this rule only applies for non-replicated nonce types (with $a = 1$): intuitively, these types can only be given to one single nonce.

This is however not the case of replicated nonce types (with $a = \infty$): we cannot know whether we are comparing the value of two copies of a nonce coming from different instances of a replicated process, or from the same instance. This case is handled by rule PIFLR'*. Although we know that the nonces on both sides are of the same type, we cannot assume that they are equal. Yet, a replicated refinement type guarantees us that any nonce with this type has been generated in the same replicated instance on the left and on the right. This means that the two nonces we compare either come from the same instance on both sides, or they come from different instances on both sides. Thus the equality check always yields the same result in the two processes.

Similarly, when the type of one of the messages being compared indicates it is a pair, while the other has a nonce type, we know that the equality check will always fail on both sides. Rule PIFI allows us to only typecheck the else-branches. This could easily be extended to more cases of incompatible types.

All these special cases highlight how a careful treatment of names in terms of equivalence classes (statically captured by types) is a powerful device to enhance the expressiveness of the analysis.

Another special case is if the messages being compared are of type HH on the left, and of type LL on the right. As a secret of high integrity can never be equal to a public value of low integrity, we know that both processes will take the else branch (PIFS). This rule is crucial, since it may allow us to prune the LL typing branch produced by asymmetric decryption.

Finally, rule PIFALL lets us typecheck any conditional by simply checking the four possible branch combinations. In contrast to the other rules for conditionals that we presented, this rule does not require any other preconditions or checks on the terms M, M', N, N'.

Remark As a final remark on the typing rules, notice that we do not have any typing rule for replication: this is in line with our general idea of typing a bounded number of sessions and then extending this result to the unbounded case in the constraint checking phase, as detailed in the next sections.

2.4 Constraints

Our type system collects constraints that intuitively correspond to (an overapproximation of) symbolic messages that the attacker may see (or deduce) during an execution of the process. Therefore, intuitively, two processes are in trace equivalence only if the collected constraints are in static equivalence for any plausible instantiation of the variables they contain.

However, checking static equivalence of symbolic frames for exactly the instantiations that correspond to a real execution may be as hard as checking trace equivalence [46]. Conversely, checking static equivalence for *all* possible instantiations may be too strong and may prevent proving equivalence of processes. Instead, we use again the abstraction of types to overapproximate the set of possible instantiations. We use the typing information gathered by our type system,

and we consider only instantiations that comply with the type. Such instantiations are said to be *well-typed*. Actually, we even restrict our attention to instantiations where variables of type LL are only replaced with deducible terms.

Hence, we define a constraint to be *consistent* if the corresponding two frames are in static equivalence for any instantiation that can be typed and produces constraints that are included in the original constraint.

Formally, we first introduce the following ingredients.

Definition 9 (Frames associated to a set of constraints)**.** *If c is a set of constraints, let $\phi_\ell(c)$ and $\phi_r(c)$ denote the frames composed of the left and right terms of the constraints respectively (in the same fixed order).*

Definition 10 (Frames associated to environments)**.** *If Γ is a typing environment, $\phi_{\mathrm{LL}}^{\Gamma}$ denotes the frame that is composed of all low confidentiality nonces and keys in Γ,* i.e. *all nonces n such that $\exists a.\ \Gamma(n) = \tau_n^{\mathrm{LL},a}$ and keys k such that $\exists T.\Gamma(k,k) <: \mathrm{key}^{\mathrm{LL}}(T)$, as well as all public encryption keys $\mathtt{pk}(k)$ and verification keys $\mathtt{vk}(k)$ for $k \in \mathrm{keys}(\Gamma)$. This intuitively corresponds to the initial knowledge of the attacker.*

Definition 11 (Notations)**.** *For a typing environment Γ, we denote by $\Gamma_{\mathcal{X}}$ its restriction to variables, and by $\Gamma_{\mathcal{N},\mathcal{K}}$ its restriction to names and bikeys.*

Definition 12 (Well-typed substitutions)**.** *Two ground substitutions σ, σ' are well-typed in Γ with constraint c_σ if they preserve the types for variables in Γ,* i.e.

- $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma') = \mathrm{dom}(\Gamma_{\mathcal{X}})$,
- *and $\forall x.\ \Gamma \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$ for some c_x such that $c = \bigcup_{x \in \mathrm{dom}(\Gamma_{\mathcal{X}})} c_x$.*

We then write $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma_{\mathcal{X}} \to c_\sigma$.

The instantiation of a constraint is defined as expected:

Definition 13 (Instantiation of constraints)**.** *If c is a set of constraints, and σ, σ' are two substitutions, let $[\![c]\!]_{\sigma,\sigma'}$ be the instantiation of c by σ on the left and σ' on the right, that is, $[\![c]\!]_{\sigma,\sigma'} = \{M\sigma \sim N\sigma' \mid M \sim N \in c\}$.*

Similarly we write for a constraint set C

$$[\![C]\!]_{\sigma,\sigma'} = \{([\![c]\!]_{\sigma,\sigma'}, \Gamma) \mid (c,\Gamma) \in C\}.$$

We can now formally define the consistency property.

Definition 14 (Consistency)**.** *A set of constraints c is* consistent *in an environment Γ if for all subsets $c' \subseteq c$ and $\Gamma' \subseteq \Gamma$ such that $\Gamma'_{\mathcal{N},\mathcal{K}} = \Gamma_{\mathcal{N},\mathcal{K}}$ and $\mathrm{vars}(c') \subseteq \mathrm{dom}(\Gamma')$, for all (ground) substitutions σ,σ' well-typed in Γ' with a constraint c_σ such that $c_\sigma \subseteq [\![c']\!]_{\sigma,\sigma'}$, the frames $\phi_{\mathrm{LL}}^{\Gamma'} \cup \phi_\ell(c')\sigma$ and $\phi_{\mathrm{LL}}^{\Gamma'} \cup \phi_r(c')\sigma'$ are statically equivalent.*

We say that (c,Γ) is consistent if c is consistent in Γ, and that a constraint set C is consistent if each element $(c,\Gamma) \in C$ is consistent.

Note that not only do we restrict our attention to instantiations σ, σ' that are well-typed, but we also require that typechecking them produces as constraints a subset of the constraints for the frames ϕ, ϕ' (once instantiated). The intuition behind this restriction is that the constraint c_ϕ should contain everything the attacker can observe in an execution that produces the frames

ϕ, ϕ'. Similarly, the constraint c_σ is intended to contain what the attacker can observe from the public messages (*i.e.* those of type LL) in σ, σ'. Intuitively, in any actual run of the protocol, the information learned from c_σ should already be contained in the information learned from c_ϕ, which is why it is sufficient to check consistency for instantiations that satisfy this condition. We will show later on that this intuition is correct in our type system.

2.5 Soundness

In this section, we provide our first main result: the soundness of our type system, *i.e.* whenever two processes can be typed with consistent constraints, then they are in trace equivalence.

Our type system soundly enforces trace equivalence: if we can typecheck P and Q then P and Q are equivalent, provided that the corresponding constraint set is consistent.

Theorem 1 (Typing implies trace equivalence)**.** *For all P, Q, and C, for all Γ containing only keys, if $\Gamma \vdash P \sim Q \to C$ and C is consistent, then $P \approx_t Q$.*

We now present the proof for Theorem 1. Rather than a fully formal proof, we choose to give an intuition of the structure of the proof, by only describing the main lemmas, and how they follow from one another. The fully detailed proofs are provided in Appendix A.1. Unless specified otherwise, the environments Γ considered in the lemmas are implicitly assumed to be well-formed.

The core of the proof is an invariant that basically states that when two processes P, Q can be typechecked with a consistent constraint, then any reduction step performed in P can be mimicked by one or several steps in Q, with the same observable action. The processes obtained after reduction can still be typechecked with consistent constraints, and the frames of output messages during the reduction can be given type LL. This invariant will be formally stated later.

Intuitively, by applying this invariant successively, we know that for such P, Q, any chain of reduction in P can be simulated in Q so that the sequences of messages on the left and on the right have type LL. We then prove that this implies these two sequences of messages are statically equivalent. This shows the trace inclusion $P \sqsubseteq_t Q$, and by symmetry we obtain the other inclusion, proving that the two processes are trace equivalent.

In the following, we present some of the main lemmas that we use to prove the invariant and the soundness theorem, explain why they are needed, and sketch their proofs somewhat informally. The fully formal proofs, and the precise statement of all lemmas, can be found in Appendix A.1.

We first establish several technical lemmas regarding the type system. These are used throughout the rest of the proofs for technical points. These technical details will not really appear in this rather high-level presentation of the proof, and hence we do not present here all these lemmas. To give an idea of the kind of properties they express, we give three examples of such lemmas. We do not however detail their proof: they are quite simple, and can be found in appendix A.1.

Lemma 1 (Typing is preserved by extending the environment)**.** *Consider processes P, Q that typecheck in Γ: $\Gamma \vdash P \sim Q \to C$. Assume Γ' contains only bindings for variables, that do not occur in* $\mathrm{dom}(\Gamma)$ *and are not bound in P or Q. Also assume Γ' does not contain union types,*

i.e. branches(Γ') = $\{\Gamma'\}$[3]. *Then P, Q still typecheck in $\Gamma \cup \Gamma'$:* $\Gamma \cup \Gamma' \vdash P \sim Q \to C'$, *where* $C' = \{(c, \Gamma_c \cup \Gamma') | (c, \Gamma_c) \in C\}$.

The proof for this lemma follows from a clear induction on the type derivation for the judgement $\Gamma \vdash P \sim Q \to C$.

Lemma 2 (Consistency for Subsets)**.**

- *If* (c, Γ) *is consistent, and* $c' \subseteq c$ *then* (c', Γ) *is consistent.*
- *Let C be a consistent constraint set. Then every subset $C' \subseteq C$ is also consistent.*
- *If* $C \cup_\forall c$ *is consistent then C also is.*

This property follows quite easily from the definition of consistency.

Lemma 3 (Substitution preserves typing)**.** *Consider messages M, N of type T in* Γ, i.e. $\Gamma \vdash M \sim N : T \to c$ *for some c. Consider two ground substitutions σ, σ' that correctly evaluate,* i.e. $\forall x.\ \sigma(x)\!\downarrow \neq \bot$ *(and similarly for σ'). Assume they are well-typed,* i.e. $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma_{\mathcal{X}} \to c_\sigma$. *Then* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma \sim N\sigma' : T \to c'$, *for some* $c' \subseteq [\![c]\!]_{\sigma,\sigma'} \cup c_\sigma$.

Here again, the proof is an easy induction on the type derivation for $\Gamma \vdash M \sim N : T \to c$.

We then establish another series of lemmas, that show the types used to describe the structure of terms (*i.e.* $(T)_{T'}$, $\{T\}_{T'}$, etc.) correctly follow the intuition we gave for them earlier.

For instance, here is (part of) the lemma for symmetric encryptions. The first point of the lemma states that messages of type $(T)_{T'}$ are indeed either the encryption of a message of type T with a message of type T', or variables (intuitively some x such that $\Gamma(x) = (T)_{T'}$). The second one states that if two messages have type $\mathtt{LL}$, and one of them is an encryption, then the other one also is. In addition, either the terms used as keys can be given a trusted key type (label $\mathtt{HH}$), and then the plaintexts have the expected payload type, or the terms used as keys as well as the plaintexts have type $\mathtt{LL}$. Intuitively, in the first case the encryption must have been produced by a legitimate process (since the key is secret), and the typechecking thus enforces that process provided a payload with the correct type. In the second case, the key is not trusted, and thus the encrypted message may have been generated by the attacker: the payload has type $\mathtt{LL}$.

Lemma 4 (Symmetric encryption types)**.** *For all* $\Gamma, T, T', M, N, M_1, M_2, c$*:*

- *If* $\Gamma \vdash M \sim N : (T)_{T'} \to c$ *then*
  - *either* $M = \mathtt{enc}(M_1, M_2)$, $N = \mathtt{enc}(N_1, N_2)$ *and* $c = c_1 \cup c_2$, *for some* M_1, M_2, N_1, N_2, c_1, c_2 *such that* $\Gamma \vdash M_1 \sim N_1 : T \to c_1$, *and* $\Gamma \vdash M_2 \sim N_2 : T' \to c_2$,
  - *or M and N are variables.*
- *If* $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : \mathtt{LL} \to c$ *then* $N = \mathtt{enc}(N_1, N_2)$ *for some* N_1, N_2, *and*
  - *either* $\Gamma \vdash M_1 \sim N_1 : T' \to c_1$, $\Gamma \vdash M_2 \sim N_2 : \mathrm{key}^{\mathtt{HH}}(T') \to c_2$, *and* $c = \{\mathtt{enc}(M_1, M_2) \sim N\} \cup c_1 \cup c_2$, *for some* T', c_1, c_2*;*
  - *or* $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$, $\Gamma \vdash M_2 \sim N_2 : \mathtt{LL} \to c_2$, *and* $c = c_1 \cup c_2$ *for some* c_1, c_2.

[3] In the full proof we actually prove a more general lemma, that does not require this condition, but we only present this particular case here, as it makes for a more legible statement.

Proof sketch. Both points are proved by induction on the typing derivation: we study all possible cases for the last rule of the derivation, and all of them are rather straightforward. For the first point for instance, this last rule can only be TEnc, TVar, or TSub, by the form of the type $(T)_{T'}$. In the TEnc and TVar cases, the premises of the rule directly prove the claim. In the TSub case, earlier technical lemmas (described in Appendix A.1) show us that before subtyping, M and N had necessarily been given type $(T'')_{T'}$ for some $T'' <: T$: applying the induction hypothesis to that judgement proves the claim. □

Armed with these technical lemmas, we are now able to prove more meaningful properties about the behaviour of messages that can be typed.

The following lemma states that the typing rules for destructors, used in rule PLet, are sound. More precisely, we establish that when a destructor is given typed T by these rules, then instantiating it in a well-typed manner produces a term that either fails to evaluate on both sides, or evaluates on both sides to messages of type T. In addition, in the second case, typing the messages produces as constraints a subset of the constraints associated with the instantiation.

Lemma 5 (Soundness of destructor rules)**.** *Consider destructors d, d', and Γ, T, c such that $\Gamma \vdash d \sim d' : T$. Let σ, σ' be ground substitutions such that $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma') = \mathrm{vars}(d) \cup \mathrm{vars}(d')$, and $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c_\sigma$, where $\Gamma' = \Gamma_{\mathcal{N},\mathcal{K}} \cup \Gamma|_{\mathrm{dom}(\sigma)}$. Then*

1. $(d\sigma)\!\downarrow = \bot \iff (d'\sigma')\!\downarrow = \bot$
2. *And if $(d\sigma)\!\downarrow \neq \bot$ then there exists $c \subseteq c_\sigma$ such that*

$$\Gamma_{\mathcal{N},\mathcal{K}} \vdash (d\sigma)\!\downarrow \sim (d'\sigma')\!\downarrow : T \to c$$

Proof sketch. We prove this property by examining, for each sort of destructor (symmetric and asymmetric decryption, signature verification, pair projection) which of the rules can lead to the judgement $\Gamma \vdash d \sim d' : T$. For instance, in the case of symmetric decryption, one possibility is that this judgement was obtained by rule DDecL. In that case we know that $T = \mathtt{LL}$, and that $d = d' = \mathtt{dec}(x, k)$ for some x, k, T such that $\Gamma(k, k) <: \mathrm{key}^{\mathtt{LL}}(T')$ and $\Gamma(x) = \mathtt{LL}$. Hence, by well-typedness, $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c$ for some $c \subseteq c_\sigma$. Lemma 4 then guarantees us that if $\sigma(x)$ is indeed an encryption with key k, then so is $\sigma'(x)$, and conversely. This proves the first point: decryption either succeeds on both sides, or fails on both sides. In addition, Lemma 4 also implies that when decryption succeeds, the plaintexts in the two encryptions have type $\mathtt{LL}$ with a constraint $c' \subseteq c \subseteq c_\sigma$. This proves the claim for that case. All the other cases have similar proofs, that use the other technical lemmas mentioned earlier. □

This property is required to establish our invariant: indeed, we need to know that after performing a reduction step in well-typed processes, we still obtain well-typed processes. In the case of a destructor application, if the processes were typechecked using rule PLet, we know that the (uninstantiated) destructor is given some type T by the destructor typing rules. We also know that the two else-branches typecheck, as well as the two then-branches when adding to the environment that the destructor produced a term of type T. This suffices to show that the two processes still typecheck after reduction only if we can be sure that the two processes follow the same branch (*i.e.*, both then-branches or both else-branches), and that when the destructor application succeeds, it indeed yields a term of type T. This is precisely what Lemma 5 tells us.

We also prove the following property. It states that when two messages can be typed (with any type), then their evaluation either succeeds for both, or fails for both. Note that messages

do not contain destructors: thus the only reason why the evaluation might fail is the use of something other than a key in a key position in the message.

Lemma 6 (Typable messages either reduce on both sides, or fail on both sides)**.** *For all (well-formed)* Γ*, for all messages* M*,* M'*, for all* T*,* c*, if*

$$\Gamma \vdash M \sim N : T \to c,$$

then

$$M{\downarrow} = \bot \iff N{\downarrow} = \bot.$$

Proof sketch. The proof is by induction on the type derivation of $\Gamma \vdash M \sim N : T \to c$.

The most interesting case is arguably that of rule TEnc: in that case $M = \mathtt{enc}(M_1, M_2)$, $N = \mathtt{enc}(N_1, N_2)$ and $T = (T_1)_{T_2}$ for some $M_1, M_2, N_1, N_2, c_1, c_2, T_1, T_2$ such that $\Gamma \vdash M_i \sim N_i : T_i \to c_i$ (for $i = 1, 2$), T_2 is either $\mathtt{LL}$ or a key type, and $c = c_1 \cup c_2$. The evaluation of M (resp. N) fails when $M_1{\downarrow} = \bot$ (resp. N_1), or when $M_2{\downarrow}$ (resp. N_2) is not a key. By induction, $M_1{\downarrow} = \bot$ if and only if $N_1{\downarrow} = \bot$. In addition, it can be proved that since $\Gamma \vdash M_2 \sim N_2 : T_2 \to c_2$ and T_2 is $\mathtt{LL}$ or a key type, M_2 and N_2 are necessarily either two keys, or none of them is a key (this property is one of the technical lemmas mentioned earlier). This proves the claim for this case. The other cases are similar, again relying on technical properties established earlier. □

As for the previous lemma, this property is required to prove our invariant. It is important when considering the output of messages. Consider processes $\mathtt{out}(M).P$, $\mathtt{out}(N).Q$ that were typechecked using rule POut. We need (among other conditions) to ensure that if the output can be performed in the first process, then it can also be performed in the second one.

The semantics specifies that a message can only be output if it successfully evaluates. We cannot now *a priori* that M and N do. It may indeed be possible that the messages are *e.g.* encrypted with a key provided by the attacker, in which case, depending on whether the attacker gave an actual key or some other message, the processes might reduce or not. This lemma guarantees us that M and N will either evaluate on both sides or fail on both sides, and thus that if the left process reduces, then so does the right process.

We next prove another key lemma, which formally confirms the intuition given earlier, that the attacker should only be able to construct messages of type $\mathtt{LL}$. More precisely, it states that from two frames of messages of type $\mathtt{LL}$ (one on the left, one on the right), by applying attacker recipes, the attacker (1) cannot construct terms that would evaluate on one side and fail on the other, (2) can only obtain messages of type $\mathtt{LL}$ (when the evaluation succeeds).

At this point we introduce a notation for such frames of type $\mathtt{LL}$, that represent the knowledge of the attacker:

Definition 15 ($\mathtt{LL}$ frames)**.** *Consider two frames* ϕ*,* ϕ' *with the same domain. We say that* ϕ*,* ϕ' *have type* $\mathtt{LL}$ *in an environment* Γ *with constraint* c*, and write* $\Gamma \vdash \phi \sim \phi' : \mathtt{LL} \to c$*, if* $\forall x \in \mathrm{dom}(\phi).\ \Gamma \vdash \phi(x) \sim \phi'(x) : \mathtt{LL} \to c_x$ *for some* c_x *such that* $c = \bigcup_{x \in \mathrm{dom}(\phi)} c_x$*.*

Formally, the aforementioned property is then:

Lemma 7 ($\mathtt{LL}$ type is preserved by attacker terms)**.** *Consider* Γ*, and two frames* ϕ *and* ϕ' *such that* $\Gamma \vdash \phi \sim \phi' : \mathtt{LL} \to c_\phi$*. Then for all attacker term* R *with* $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi)$*, either*

$$\Gamma \vdash R\phi{\downarrow} \sim R\phi'{\downarrow} : \mathtt{LL} \to c' \text{ for some } c \subseteq c_\phi$$

or

$$R\phi{\downarrow} = R\phi'{\downarrow} = \bot.$$

Proof sketch. This proof is by induction on the attacker recipe R. We describe here the case where $R = \mathtt{dec}(S, K)$ for some recipes S, K. The other cases are mostly similar. We apply the induction hypothesis to K, and distinguish three cases.

1. if $K\phi\downarrow = \bot$, then by induction $K\phi'\downarrow = \bot$, and thus $R\phi\downarrow = R\phi'\downarrow = \bot$.

2. otherwise, by induction, $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to c$ for some $c \subseteq c_\phi$. Then:

   - either $K\phi\downarrow$ is not a key, and then we can show that $K\phi'\downarrow$ cannot be a key either (this is one of the technical lemmas, provided in Appendix A.1). Thus, again, $R\phi\downarrow = R\phi'\downarrow = \bot$.
   - or $K\phi\downarrow$ is a key k, and the same technical lemma shows that $K\phi'\downarrow$ is necessarily the same key k, and that either $\Gamma(k,k) <: \mathrm{key}^{\mathtt{LL}}(T)$ (for some T), or $k \in \mathcal{FK}$ is a key produced by the attacker. We then apply the induction hypothesis to S: either $S\phi\downarrow = S\phi'\downarrow = \bot$, or $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c'$ for some $c' \subseteq c_\phi$. In the first case, again, $R\phi\downarrow = R\phi'\downarrow = \bot$ and the claim holds. In the second case, we apply Lemma 4:
     - either $S\phi\downarrow$ and $S\phi'\downarrow$ are both encryptions with k: *i.e.* $S\phi\downarrow = \mathtt{enc}(M,k)$ and $S\phi'\downarrow = \mathtt{enc}(N,k)$ for some M, N, and Lemma 4 also tells us that $\Gamma \vdash M \sim N : \mathtt{LL} \to c''$ for some $c'' \subseteq c' \subseteq c_\phi$. Since in this case $R\phi\downarrow = M$ and $R\phi'\downarrow = N$, this proves the claim.
     - or neither of them are: then again, $R\phi\downarrow = R\phi'\downarrow = \bot$ and the claim holds.

□

This property validates the fact that we assign type LL to messages input from the network. It is crucial in order to prove the invariant, in particular in the case where we consider processes of the form $\mathtt{in}(x).P$, $\mathtt{in}(x).Q$ (rule PIN). Indeed, these will reduce to P and Q respectively, with some message constructed by the attacker using the frame of previous outputs being stored in variable x. To show that P and Q still typecheck, we will want to use the premise of rule PIN: it guarantees that they typecheck provided x is considered to have type LL. We know that the message actually stored into x has this type thanks to Lemma 7.

As mentioned, the invariant we prove will guarantee that for processes that typecheck with consistent constraints, any execution produces frames that are of type LL, with consistent constraints. For this to imply trace equivalence, we need to show that such frames are statically equivalent.

As we explained earlier, the intuition is that the constraints produced when typechecking messages with type LL contain all the information the attacker learns when observing these messages. Static equivalence of the constraints, which is intuitively what consistency entails, then implies the frames to be statically equivalent.

More precisely, we first show the following property, which formalises that intuition. It states that messages of type LL can always be reconstructed by the attacker using only the messages contained in the constraints, as well as public names and keys.

Lemma 8 (LL terms are recipes on their constraints)**.** *Consider ground messages M, N, such that $\Gamma \vdash M \sim N : \mathtt{LL} \to c$ for some Γ, c. Then there exists an attacker recipe R without destructors such that $M = R(\phi_\ell(c) \cup \phi^{\Gamma}_{\mathtt{LL}})$ and $N = R(\phi_r(c) \cup \phi^{\Gamma}_{\mathtt{LL}})$.*

Proof sketch. We actually prove an equivalent formulation: we show that this holds as soon as messages have any type $T <: \mathtt{LL}$. That statement is equivalent to the one above, thanks to rule TSub, but it allows us to prove the claim by induction on the typing derivation for $\Gamma \vdash M \sim N : T \to c$ (the first formulation is indeed not inductive in the case of rule TSub). We study each possible case for the last rule applied in that derivation.

Most cases are rather straightforward, either directly or just by applying the induction hypothesis. For instance, in the case of rules TEncH or THash, $M \sim N$ is directly added to the constraint, which makes the claim trivial. For rules such as TPair or THashL, c contains the constraints generated when typing each component of the pair (resp. the message being hashed). By induction, these components can therefore be constructed by the attacker by applying some recipe on (a subset) of the constraints. Then M and N themselves can be obtained by pairing (resp. hashing) these recipes.

Some cases are slightly more involved and require the use of some technical lemmas. We will not detail all cases here, but only, for the sake of example, that of rule TEncL. In that case, we know by the premise of the rule that $\Gamma \vdash M \sim N : (\mathtt{LL})_{T'} \to c$ with $T' <: \mathrm{key}^{\mathtt{LL}}(T'')$ (for some T''). By Lemma 4, we then know that M, N are encryptions of messages M', N' of type $\mathtt{LL}$ (with some constraint $c' \subseteq c$) with some untrusted key. By induction[4], we then know that M', N' can be constructed by the attacker using the constraint c. Since the attacker knows the untrusted encryption key, he can thus also construct M, N, which prove the claim in that case. □

Using that lemma, we can then prove that frames of type $\mathtt{LL}$ with consistent constraints are statically equivalent.

Lemma 9 ($\mathtt{LL}$ frames with consistent constraints are statically equivalent)**.** *For all ground frames ϕ, ϕ' such that $\Gamma \vdash \phi \sim \phi' : \mathtt{LL} \to c$ for some c consistent in $\Gamma_{\mathcal{N},\mathcal{K}}$, ϕ and ϕ' are statically equivalent.*

Proof. Consider two attacker recipes R, R', that use variables in $\mathrm{dom}(\phi)(= \mathrm{dom}(\phi'))$. Since $\Gamma \vdash \phi \sim \phi' : \mathtt{LL} \to c$, by definition for any $x \in \mathrm{dom}(\phi)$ $\phi(x)$ and $\phi'(x)$ have type $\mathtt{LL}$ in Γ for some subset of c. Thus, by Lemma 8, there exists a recipe R_x such that $\phi(x) = R_x(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}})$ and $\phi'(x) = R_x(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$. By replacing in R and R' each occurrence of every such variable x with R_x, we obtain two attacker recipes $\overline{R}$, $\overline{R'}$, such that $R\phi = \overline{R}(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}})$, $R\phi' = \overline{R}(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$, $R'\phi = \overline{R'}(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}})$, and $R'\phi' = \overline{R'}(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$. Since ϕ, ϕ' are ground, it is easy to see from the typing rules for messages that so is c. Thus, by definition of consistency, $\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}}$ and $\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}}$ are statically equivalent. Hence

$$\overline{R}(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}}) = \overline{R'}(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}}) \Leftrightarrow \overline{R}(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}}) = \overline{R'}(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$$

i.e.

$$R\phi = R'\phi \Leftrightarrow R\phi' = R'\phi'$$

which proves that ϕ, ϕ' are statically equivalent. □

Note that this lemma only applies to ground frames, and not to symbolic frames where variables remain uninstantiated. According to the semantics of pi-calculus, when a process

[4] Applying the induction hypothesis requires us to show that the type derivation for $\Gamma \vdash M' \sim N' : \mathtt{LL} \to c'$ is shorter that the one for $\Gamma \vdash M \sim N : \mathtt{LL} \to c$. This is done in the complete proof by actually proving a stronger version of Lemma 4, that does not only guarantee the existence of this derivation, but also that it is shorter than the one for $\Gamma \vdash M \sim N : (\mathtt{LL})_{T'} \to c$.

reduces, a symbolic frame ϕ is produced, as well as a ground substitution σ that contains the instantiation of each variable in ϕ for the current execution. We will then apply the previous lemma to the ground frame $\phi\sigma$. The invariant that follows lets us know that this frame (together with the corresponding frame from the right process) has type $\mathtt{LL}$, with consistent constraints.

Let us now state formally the invariant. For legibility, we present here a slightly simplified statement, where some conditions regarding the domains of the substitutions and the set of variables bound in processes are omitted. The complete statement can be found in Appendix A.1.

Lemma 10 (Invariant)**.** *Consider an environment Γ, and multisets of processes $\mathcal{P} = \{P_i\}_i$, $\mathcal{Q} = \{Q_i\}_i$ with $|\mathcal{P}| = |\mathcal{Q}|$ such that $\forall i.\Gamma \vdash P_i \sim Q_i \to C_i$ for some C_i. Also consider frames ϕ_P, ϕ_Q such that $\Gamma \vdash \phi_P \sim \phi_Q : \mathtt{LL} \to c_\phi$ for some c_ϕ. Let σ_P, σ_Q be ground substitutions such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$ for some c_σ. Assume $c_\sigma \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$, and that $[\![(\cup_{\times i} C_i) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is consistent. Consider a reduction step on the left: $(\mathcal{P}, \phi_P, \sigma_P) \xrightarrow{\alpha} (\mathcal{P}', \phi'_P, \sigma'_P)$.*

Then there exists a sequence of actions $w =_\tau \alpha$, such that $(\mathcal{Q}, \phi_Q, \sigma_Q) \xrightarrow{w}_* (\mathcal{Q}', \phi'_Q, \sigma_Q)$ for some $\mathcal{Q}'$, ϕ'_Q, σ'_Q, with $|\mathcal{Q}'| = |\mathcal{P}'|$ and $\forall i.\Gamma' \vdash P'_i \sim Q'_i \to C'_i$ for some Γ', C'_i. In addition $\Gamma' \vdash \phi'_P \sim \phi'_Q : \mathtt{LL} \to c'_\phi$ and $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma'_P \sim \sigma'_Q : \Gamma'_{\mathcal{X}} \to c'_\sigma$ for some c'_σ, c'_ϕ such that $c'_\sigma \subseteq \left[\!\!\left[c'_\phi\right]\!\!\right]_{\sigma'_P,\sigma'_Q}$ and $\left[\!\!\left[(\cup_{\times i} C'_i) \cup_\forall c'_\phi\right]\!\!\right]_{\sigma'_P,\sigma'_Q}$ is consistent.

Proof sketch. Basically, according to the semantics of pi-calculus, the reduction step performed on $\mathcal{P}$, *i.e.* $(\mathcal{P}, \phi_P, \sigma_P) \xrightarrow{\alpha} (\mathcal{P}', \phi'_P, \sigma'_P)$, is more precisely affecting only one of the processes $P_i \in \mathcal{P}$: outputting or inputting a message, performing a conditional branching, etc. is only done in one of the processes at a time.

The proof then consists in a case disjunction on the last typing rule applied in the type derivation for P_i, *i.e.* $\Gamma \vdash P_i \sim Q_i \to C_i$.

We do not give all details here, but rather explain in general how the proof goes. In each case, the proof has the following structure.

- The typing rule imposes some conditions on the form of processes P_i, Q_i: *e.g.* if this rule is POut then $P_i = \mathtt{out}(M).P'_i$ (and similarly for Q_i), if the rule is PIn then $P_i = \mathtt{in}(x).P'_i$, and so on.

- In turn, the form of P_i gives us information about which reduction rule is applied to P_i in the reduction step. In the cases of POut and PIn for instance, it can only be rule Out (resp. In).

- Since the typing rules require that Q_i has the same head symbol as P_i, the same reduction rule can be applied to Q_i. Depending on the case, additional conditions may need to be checked. For instance, in the case of an output, we need to make sure that the message in Q_i is legal to output, knowing that the one in P_i is. That is, we must show that that message successfully evaluates. As explained earlier, this is done using Lemma 6. In other cases we similarly use other lemmas that were omitted here.

- This shows that $\mathcal{Q}$ can be reduced too, with the same actions as $\mathcal{P}$ (in some cases, silent actions may need to be performed before actually performing in $\mathcal{Q}$ the same observable action as in $\mathcal{P}$; we omit such details here).

- We then need to show the conditions on the type of processes, frames and substitutions hold after the reduction. Most of the processes are unchanged and still typecheck, although the environment may have been extended with new variables, *e.g.* in the input case: in such cases we use Lemma 1. For the processes actually being reduced, these conditions usually follow from the premises of the typing rule. For instance in the output case, one premise of the rule guarantees us that the continuation processes after the output still typecheck. The conditions on the frames is that they should have type LL: the only case where some messages are added to the frame is the output case, in which the premises of rule POut enforce this condition. Regarding the substitutions, what we must show is basically that the messages actually stored in variables in the execution have the type the typing rules assign to these variables. For instance in the input case, the variable in which the message is stored is supposed to have type LL. We thus need to show the messages actually input have this type: as explained earlier, this is done thanks to Lemma 7. In the case of a destructor application, we similarly use Lemma 5; other cases use some lemmas we omitted in this presentation.

- Finally, we need to prove the consistency of the constraints after reduction. More precisely, we know that $[\![(\cup_{\times i} C_i) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is consistent and must show that $\left[\!\!\left[(\cup_{\times i} C'_i) \cup_\forall c'_\phi\right]\!\!\right]_{\sigma'_P,\sigma'_Q}$ is. Basically, what may happen during a reduction step is that some constraints disappear from the constraints C_i of the processes, and are added to the constraint c'_ϕ of the frames. That is typically the case when the reduction step is an output: typing the continuation processes after the output no longer generates constraints for the messages output, however these messages are added to the frames, and thus these constraints are instead generated when typing the frames. We thus use some technical lemmas regarding consistency and union and instantiation of constraints, such as Lemma 2 to prove consistency of the new constraints. □

The intuition behind the constraint $C = [\![(\cup_{\times i} C_i) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ may be a bit difficult to get. Before we get to the remainder of the proof, let us give an informal description of the role this constraint plays, omitting all technical details, just to explain the intuition. One way to do this is to examine how C evolves during a reduction. Basically, in the beginning, the frames and substitutions are empty, and C is thus simply the constraint set for the initial processes. In particular, it contains the constraints for all (uninstantiated) messages that will be output during the reduction. During the reduction, each time a message is output, it is added to ϕ. Conversely the output operation disappears from the process remaining to be reduced, as it has been executed. The constraints for the message, thus initially present in $(\cup_{\times i} C_i)$, are hence "passed" to c_ϕ. Thus, intuitively, $(\cup_{\times i} C_i) \cup_\forall c_\phi$ as a whole does not really change, which is basically what allows us to prove in the invariant that its consistency is preserved. Once the end of the execution has been reached, intuitively, no processes are left to reduce, and thus C only consists in the instantiation of c_ϕ. The invariant thus lets us know us that $[\![c_\phi]\!]_{\sigma_P,\sigma_Q}$ is consistent. As discussed earlier, we will need to know that the instantiated frames $\phi_P\sigma_P$, $\phi_Q\sigma_Q$ have consistent constraints (in order to show their static equivalence). By Lemma 3, this constraint is basically $c_\sigma \cup [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$: as the invariant also establishes that $c_\sigma \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$, its consistency follows from the previous observation.

Note that we show, as part of the invariant, that the constraints c_σ from the instantiations σ, σ' are contained in the (instantiated) constraints $[\![c_\phi]\!]_{\sigma,\sigma'}$ of the frames. This justifies the

intuition presented in Section 2.4 that considering only such instantiations in the definition of consistency is sufficient.

Using the invariant, together with Lemma 9, we can now prove that typing implies trace inclusion:

Lemma 11 (Typing implies trace inclusion)**.** *For all P, Q, and C, for all Γ containing only keys, if $\Gamma \vdash P \sim Q \to C$ and C is consistent, then $P \sqsubseteq_t Q$.*

Proof sketch. Consider a sequence of reductions $(\{P\}, \emptyset, \emptyset) \xrightarrow{s}_* (\mathcal{P}, \phi_P, \sigma_P)$ on the left process. By applying successively the invariant (Lemma 10), we can construct a sequence of reductions $(\{Q\}, \emptyset, \emptyset) \xrightarrow{s'}_* (\mathcal{Q}, \phi_Q, \sigma_Q)$ on the right process. To prove the claim, it is then sufficient to show that $\phi_P\sigma_P$ and $\phi_Q\sigma_Q$ are statically equivalent. First, by the invariant, we know that $\Gamma \vdash \phi_P \sim \phi_Q : \mathtt{LL} \to c_\phi$ and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$ for some Γ, c_ϕ, c_σ. Using Lemma 3, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \phi_P\sigma_P \sim \phi_Q\sigma_Q : \mathtt{LL} \to c$ for some $c \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q} \cup c_\sigma$, which is itself a subset of $[\![c_\phi]\!]_{\sigma_P,\sigma_Q}$ by the invariant. Lemma 9 will conclude the proof if we can show that c is consistent in $\Gamma_{\mathcal{N},\mathcal{K}}$. By the invariant, $[\![C \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is consistent, where C is the constraint obtained when typechecking processes in $\mathcal{P}$, $\mathcal{Q}$. Using technical lemmas analogous to Lemma 2, we can prove this implies the consistency of $[\![c_\phi]\!]_{\sigma_P,\sigma_Q}$, which (again using technical lemmas) implies the consistency of c. □

It is clear from the typing rules that the type system is symmetric, in the sense that if $\Gamma \vdash P \sim Q \to C$, then $\overline{\Gamma} \vdash Q \sim P \to \overline{C}$, where $\overline{\Gamma}$ and $\overline{C}$ are obtained from Γ and C by exchanging the left and right terms in constraints, as well as the left and right types in refinement types, and the left and right keys in bikeys. It is clear from the definition of consistency that consistency of C and $\overline{C}$ are equivalent.

Therefore, by symmetry, the soundness theorem (Theorem 1) follows from Lemma 11.

Since we do not have typing rules for replication, Theorem 1 only allows us to prove equivalence of protocols for a *finite* number of sessions. Thanks to our infinite nonce types, we can however still prove equivalence for an *unbounded* number of sessions, as detailed in the next section.

2.6 From bounded to unbounded number of sessions

In this section, we show how we can lift the soundness result from the previous section to the case of replicated processes, *i.e.* with an unbounded number of section, while still only needing to typecheck a finite number of sessions.

For more clarity, without loss of generality we consider that for each infinite nonce type $\tau_m^{l,\infty}$ appearing in the processes, the set of names $\mathcal{BN}$ also contains an infinite number of fresh names $\{m_i \mid i \in \mathbb{N}\}$ which do not appear in the processes or environments. We will denote by $\mathcal{N}_0$ the set of unindexed names in $\mathcal{BN}$ and by $\mathcal{N}_i$ the set of indexed names. We similarly assume that for all the variables x appearing in the processes, the set $\mathcal{X}$ of all variables also contains fresh variables $\{x_i \mid i \in \mathbb{N}\}$ which do not appear in the processes or environments. We denote $\mathcal{X}_0$ the set of unindexed variables, and $\mathcal{X}_i$ the set of indexed variables. Finally, we assume for all key k declared in the processes with type $\mathrm{seskey}^{l,\infty}(T)$ that the set $\mathcal{BK}$ contains keys $\{k_i \mid i \in \mathbb{N}\}$. This is simply a naming convention, and we do not lose any generality.

Intuitively, whenever we can typecheck a process of the form $\texttt{new}\ n : \tau_n^{l,1}.\ \texttt{new}\ m : \tau_m^{l,\infty}.\ P$, with the rules presented in Section 2.3.4, we can actually also typecheck

$$\texttt{new}\ n : \tau_n^{l,1}.\ (\texttt{new}\ m_1 : \tau_{m_1}^{l,1}.P_1 \mid \ \ldots\ \mid \texttt{new}\ m_k : \tau_{m_k}^{l,1}.P_k)$$

where in P_i, the nonce m has been renamed to m_i and variables x have been renamed to x_i. That is, if we can typecheck a process representing a single session of a protocol, then we can also typecheck any number of copies of this process in parallel, even if new nonces are generated within the replicated process, provided they are given a nonce type with $a = \infty$.

We will now first define formally this operation of expanding a process to n sessions, and then formally state the claim above.

2.6.1 Definitions: expansion to n sessions

Formally, we define the renaming of a term t for session i as follows.

Definition 16 (Renaming of a term)**.** *We denote by* $[\,t\,]_i^\Gamma$*, the term* t *in which names* n *such that* $\Gamma(n) = \tau_n^{l,\infty}$ *for some* l *are replaced with* n_i*, keys* k *such that* $\Gamma(k,k) = \text{seskey}^{l,\infty}(T)$ *for some* l*,* T *are replaced with* k_i*, and variables* x *are replaced with* x_i*.*

Similarly, when a message is of type $[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$, it intuitively represent a value generated in a replicated process. As briefly mentioned earlier, the intended meaning of that infinite refinement type is that when actually expanding the replication to n copies of the process, the type could represent values generated in any of the copies, as long as they are created in the same copy on the left and on the right. That is, values of type $[\![\tau_{m_i}^{l,1}\,;\,\tau_{p_i}^{l',1}]\!]$, for any i.

The nonce type $\tau_m^{l,\infty}$ represents infinitely many nonces (one for each potential session). That is, for n sessions, the type $[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$ is intended to represent all $[\![\tau_{m_i}^{l,1}\,;\,\tau_{p_i}^{l',1}]\!]$.

Formally, given a type T, we define its expansion to n sessions, denoted $[\,T\,]^n$, as follows.

Definition 17 (Expansion of a type)**.**

$$\begin{aligned}
[\,l\,]^n &= l \\
[\,T * T'\,]^n &= [\,T\,]^n * [\,T'\,]^n \\
\left[\,\text{key}^l(T)\,\right]^n &= \text{key}^l([\,T\,]^n) \\
\left[\,\text{eqkey}^l(T)\,\right]^n &= \text{eqkey}^l([\,T\,]^n) \\
\left[\,\text{seskey}^{l,a}(T)\,\right]^n &= \text{seskey}^{l,1}([\,T\,]^n) \\
[\,\text{pkey}(T)\,]^n &= \text{pkey}([\,T\,]^n) \\
[\,\text{vkey}(T)\,]^n &= \text{vkey}([\,T\,]^n) \\
[\,(T)_{T'}\,]^n &= ([\,T\,]^n)_{[\,T'\,]^n} \\
[\,\{T\}_{T'}\,]^n &= \{[\,T\,]^n\}_{[\,T'\,]^n} \\
[\,T \vee T'\,]^n &= [\,T\,]^n \vee [\,T'\,]^n \\
\left[\,[\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!]\,\right]^n &= [\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!] \\
\left[\,[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]\,\right]^n &= \textstyle\bigvee_{j=1}^n [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]
\end{aligned}$$

where $l, l' \in \{\texttt{LL}, \texttt{HH}, \texttt{HL}\}$*,* $k \in \mathcal{K}$*.*

The crucial point is the last one: infinite refinement types for nonces $\left[\!\left[\tau_m^{l,\infty} ; \tau_p^{l',\infty} \right]\!\right]^n$ are expanded into a union type $\bigvee_{j=1}^{n} [\![\tau_{m_j}^{l,1} ; \tau_{p_j}^{l',1}]\!]$. We may also note that, contrary to infinite nonce refinement types, infinite session key types are not turned into union types. The reason for this is that the refinement types carry information regarding the value of the messages they can be given to. When expanding the replicated copies of a process, there are now several possibilities for the value of a message that had previously an infinite refinement type: it could be the value generated in any of the n copies. Hence the need for a union type. However, session key types do not carry information about the value of the key, but only about the type of messages it should encrypt or sign. This information does not change once the key may have several values, from different copies of the process: thus there is no need for a union type here.

Note finally that the size of the expanded type $[\, T \,]^n$ depends on n.

We need to adapt typing environments accordingly.

Definition 18 (Renaming and expansion of typing environments)**.** *For any typing environment* Γ*, we define its renaming for session* i *as:*

$$\begin{aligned}
[\, \Gamma \,]_i = \{ & x_i : T \mid \Gamma(x) = T \} \\
& \cup \{ (k, k') : T \mid \Gamma(k, k') = T \wedge \forall l, T'. T \neq \mathrm{seskey}^{l,\infty}(T') \} \\
& \cup \{ (k_i, k_i) : \mathrm{seskey}^{l,1}(T) \mid \Gamma(k, k) = \mathrm{seskey}^{l,\infty}(T) \\
& \cup \ \{ m : \tau_m^{l,1} \mid \Gamma(m) = \tau_m^{l,1} \} \\
& \cup \ \{ m_i : \tau_{m_i}^{l,1} \mid \Gamma(m) = \tau_m^{l,\infty} \}.
\end{aligned}$$

and then its expansion to n *sessions as*

$$\begin{aligned}
[\, \Gamma \,]_i^n = \{ & x_i : [\, T \,]^n \mid [\, \Gamma \,]_i(x_i) = T \} \cup \{ (k, k') : [\, T \,]^n \mid [\, \Gamma \,]_i(k, k') = T \} \\
& \cup \ \{ m : \tau_m^{l,1} \mid [\, \Gamma \,]_i(m) = \tau_m^{l,1} \}.
\end{aligned}$$

The idea is that $[\, \Gamma \,]_i$ is simply renaming all variables, names, keys by indexing them with the session number i, and giving them the corresponding finite nonce or key type. However renaming does not expand any types: the types of variables, or the payload types in key types may still contain infinite refinement types. The expansion $[\, \Gamma \,]_i^n$ then replaces these with their expansion, *i.e.* replacing all refinements with a union of n finite types, for each of the n sessions. Note that in $[\, \Gamma \,]_i^n$, due to the expansion, the size of the types (potentially) depends on n.

By construction, the environments contained in the constraints generated by typing do not contain union types (see rule PZero). However, refinement types with infinite nonce types introduce union types when expanded. In order to recover environments without union types after expanding, which, as we will explain in the next subsection, is needed for our consistency checking procedure, we will use the branches$(\cdot)$ operation defined in Section 2.3.2.

We define the renaming of a process for session i in a similar way.

Definition 19 (Renaming of a process)**.** *For all process* P*, for all* $i \in \mathbb{N}$*, for all environment* Γ*,* $[\, P \,]_i^\Gamma$ *as the process obtained from* P *by:*

- *for each nonce* n *declared in* P *by* `new` $n : \tau_n^{l,\infty}$*, and each nonce* n *such that* $\Gamma(n) = \tau_n^{l,\infty}$ *for some* l*, replacing every occurrence of* n *with* n_i*, and the declaration* `new` $n : \tau_n^{l,\infty}$ *with* `new` $n_i : \tau_{n_i}^{l,1}$*;*

- *for each key k declared in P by* $\texttt{new}\ k : \text{seskey}^{l,\infty}(T)$*, and each key k such that* $\Gamma(k,k) = \text{seskey}^{l,\infty}(T)$ *for some l, T replacing every occurrence of k with k_i, and the declaration* $\texttt{new}\ k : \text{seskey}^{l,\infty}(T)$ *with* $\texttt{new}\ k_i : \text{seskey}^{l,1}([\,T\,]^n)$*;*
- *replacing every occurence of a variable x with x_i.*

Finally, when typechecking two processes containing nonces with infinite nonce types, we collect constraints that represent "families" of constraints. That is, the constraints we collect represent all the constraints that would correspond to typechecking each replicated copy of the process. We therefore also need to expand them.

Definition 20 (Renaming and expansion of constraints)**.** *Given a set of constraints c, and an environment Γ, we define the renaming of c for session i in Γ as*

$$[\,c\,]_i^\Gamma = \{[\,u\,]_i^\Gamma \sim [\,v\,]_i^\Gamma \mid u \sim v \in c\}.$$

This is propagated to constraint sets as follows: the renaming of C for session i is

$$[\,C\,]_i = \{([\,c\,]_i^\Gamma, [\,\Gamma\,]_i) \mid (c,\Gamma) \in C\}$$

and its expansion to n sessions is

$$[\,C\,]_i^n = \{([\,c\,]_i^\Gamma, \Gamma') \mid \exists \Gamma.\ (c,\Gamma) \in C \ \wedge\ \Gamma' \in \text{branches}([\,\Gamma\,]_i^n)\}.$$

Again, note that the size of $[\,C\,]_i$ does not depend on the number of sessions considered, while the size of the types present in $[\,C\,]_i^n$ does. For example, for

$$C = \{(\{\mathtt{h}(x) \sim \mathtt{h}(x)\}, [x : [\![\tau_m^{\mathtt{HH},\infty}\,;\,\tau_p^{\mathtt{HH},\infty}]\!]])\},$$

we have

$$[\,C\,]_i = \{(\{\mathtt{h}(x_i) \sim \mathtt{h}(x_i)\}, [x_i : [\![\tau_m^{\mathtt{HH},\infty}\,;\,\tau_p^{\mathtt{HH},\infty}]\!]])\}$$

and

$$[\,C\,]_i^n = \{(\{\mathtt{h}(x_i) \sim \mathtt{h}(x_i)\}, [x_i : \bigvee_{j=1}^{n} [\![\tau_{m_j}^{\mathtt{HH},1}\,;\,\tau_{p_j}^{\mathtt{HH},1}]\!]])\}.$$

We now have all the notations required to formally state the previous claim, that typechecking one session is sufficient to know that any number of sessions typecheck.

2.6.2 Soundness for replicated processes

Our type system is sound for replicated processes provided that the collected constraint sets are consistent, when instantiated with all possible instantiations of the nonces and keys. In fact, we can even typecheck processes with a replicated part and a non-replicated part.

Theorem 2. *Consider processes P, Q, P' ,Q' such that P, Q and P', Q' do not share any variable. Consider Γ, containing only keys and nonces with finite key or nonce types. Assume that P and Q only bind nonces and keys with infinite nonce types,* i.e. *using* $\texttt{new}\ m : \tau_m^{l,\infty}$ *and* $\texttt{new}\ k : \text{seskey}^{l,\infty}(T)$ *for some label l and type T; while P' and Q' only bind nonces and keys with finite types,* i.e. *using* $\texttt{new}\ m : \tau_m^{l,1}$ *and* $\texttt{new}\ k : \text{seskey}^{l,1}(T)$*.*

Let us abbreviate by $\texttt{new}\ \overline{n}$ *the sequence of declarations of each nonce $m \in \text{dom}(\Gamma)$ and session key k such that $\Gamma(k,k) = \text{seskey}^{l,1}(T)$ for some l, T.*

If

- $\Gamma \vdash P \sim Q \to C$,
- $\Gamma \vdash P' \sim Q' \to C'$,
- $C' \cup_\times (\cup_{\times 1 \le i \le n} [\, C \,]_i^n)$ *is consistent for all* $n \ge 1$,

then

$$\texttt{new}\ \overline{n}.\ ((!P) \mid P') \approx_t \texttt{new}\ \overline{n}.\ ((!Q) \mid Q').$$

Basically, if P, Q typecheck with C and P', Q' with C', assuming finite and infinite nonce types are used as expected, then $P'|!P$ and $Q'|!Q$ are trace equivalent, provided the constraints formed of C' and any number of (renamed) copies of C are all consistent.

We provide in later sections a procedure to prove that such constraints of unbounded size are consistent.

For now, we present the main steps of the proof of Theorem 2. As for the proof of soundness in the non-replicated case, we do not give here the detailed proof, but rather state the main lemmas and briefly sketch their proofs. All detailed proofs can be found in Appendix A.2.

We first show that if two messages can be given type T in Γ, then the renaming of these messages for session i has type $[\, T \,]^n$ in (any branch of) the expansion of Γ to n sessions.

Lemma 12 (Typing terms with replicated names)**.** *Consider messages* M, N *such that* $\Gamma \vdash M \sim N : T \to c$ *for some* Γ, T, c. *Then for all* $n \in \mathbb{N}$ *and all* $i \in [\![1, n]\!]$, *for all* $\Gamma' \in \text{branches}([\, \Gamma \,]_i^n)$,

$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : [\, T \,]^n \to [\, c \,]_i^\Gamma$$

Proof sketch. This property is proved by induction on the type derivation Π for $\Gamma \vdash M \sim N : T \to c$. We study each possibility regarding the last typing rule applied in this derivation.

In most cases, the claim directly follows from the definition of the expanded types, environments and constraints, and the induction hypothesis. For instance in the case of rule TNonce, the form of the rule entails that M, N are two names m, n, and that $T \in \{\texttt{HH}, \texttt{HL}\}$. The derivation is then

$$\Pi = \frac{\Gamma(m) = \tau_m^{l,a} \qquad \Gamma(p) = \tau_p^{l,a}}{\Gamma \vdash m \sim p : l \to \emptyset}.$$

It is clear from the definition of $[\, \Gamma \,]_i^n$ that $\Gamma'([\, m \,]_i^\Gamma) = \tau^{l,1}_{[\, m \,]_i^\Gamma}$, and that $\Gamma'([\, p \,]_i^\Gamma) = \tau^{l,1}_{[\, p \,]_i^\Gamma}$. Then, by rule TNonce, we have $\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : l \to \emptyset$ which proves the claim for this case.

Another example would be the case of rule TEncH: then $T = \texttt{LL}$ and there exist T', k, c' such that

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash M \sim N : (T')_{T''} \to c'} \qquad T'' <: \text{key}^{\texttt{HH}}(T')}{\Gamma \vdash M \sim N : \texttt{LL} \to c = c' \cup \{M \sim N\}}.$$

By applying the induction hypothesis to Π', since $[\, (T')_{T''} \,]^n = ([\, T' \,]^n)_{[\, T'' \,]^n}$, there exists a proof Π'' of $\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : ([\, T' \,]^n)_{[\, T'' \,]^n} \to [\, c' \,]_i^\Gamma$

In addition it is clear by the definitions of $[\, \cdot \,]^n$ and of subtyping that since $T'' <: \text{key}^{\texttt{HH}}(T')$, we have $[\, T'' \,]^n <: \text{key}^{\texttt{HH}}([\, T' \,]^n)$. Therefore by rule TEncH, we have

$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : \texttt{LL} \to [\, c' \,]_i^\Gamma \cup \{[\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma\} = [\, c \,]_i^\Gamma$$

Let us finally describe one last case, which is a bit more involved: the case of rule TLR'. In that case there exist m, p, l such that $T = l$, and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash M \sim N : [\![\tau_m^{l,a}\,;\,\tau_p^{l,a}]\!] \to c}}{\Gamma \vdash M \sim N : l \to c}.$$

Let us distinguish the case where a is 1 from the case where a is ∞.

- If a is 1: by applying the induction hypothesis to Π', since $\left[\!\left[[\![\tau_m^{l,a}\,;\,\tau_p^{l,a}]\!] \right]\!\right]^n = [\![\tau_m^{l,1}\,;\,\tau_p^{l,1}]\!]$, we have
$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : [\![\tau_m^{l,a}\,;\,\tau_p^{l,a}]\!] \to [\, c \,]_i^\Gamma.$$
Thus by rule TLR', we have $[\, \Gamma \,]_i^n \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : l \to [\, c \,]_i^\Gamma$.
- If a is ∞: by applying the induction hypothesis to Π', since
$$\left[\!\left[[\![\tau_m^{l,a}\,;\,\tau_p^{l,a}]\!] \right]\!\right]^n = \bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l,1}]\!],$$
we have
$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : \bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l,1}]\!] \to [\, c \,]_i^\Gamma.$$
We can prove[5] that this implies the existence of some $j \in [\![1, n]\!]$ such that
$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l,1}]\!] \to [\, c \,]_i^\Gamma.$$
Thus, by rule TLR', we have $\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : l \to [\, c \,]_i^\Gamma$, which proves the claim. □

We also prove the following lemma, that is the analogous property to Lemma 12 for the destructor rules used by rule PLET:

Lemma 13 (Typing destructors with replicated names)**.** *For all Γ, t, t', T, if*

$$\Gamma \vdash t \sim t' : T$$

then for all $i, n \in \mathbb{N}$ such that $1 \le i \le n$,

$$[\, \Gamma \,]_i^n \vdash [\, t \,]_i^\Gamma \sim [\, t' \,]_i^\Gamma : [\, T \,]^n$$

The proof of this property is immediate by examining the typing rules for destructors.

Using these last two lemmas, as well as some additional technical lemmas omitted here, we then prove the claim made in the beginning of this section, that typechecking processes with replicated types is sufficient to ensure any renamings of these processes still typecheck when expanding the environment to n sessions.

Lemma 14 (Typing processes with expanded types)**.** *Consider processes P, Q such that $\Gamma \vdash P \sim Q \to C$ for some Γ, C. Then for all $n \in \mathbb{N}$ and $i \in [\![1, n]\!]$, there exists $C' \subseteq [\, C \,]_i^n$ such that*

$$[\, \Gamma \,]_i^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C'$$

[5] This is again using some of the technical lemmas established when proving the first soundness theorem

Proof sketch. This proof is by induction on the derivation Π of $\Gamma \vdash P \sim Q \to C$. We study each case for the last rule applied in this derivation. We will not detail all cases here, as they are all rather similar. Basically, we show that, in each case, as the renamed process have the same structure as the original ones, the same typing rule can be applied to typecheck them in the expanded environment. To do this, we need to show that the premises of the rule still hold. When the premise involves typechecking processes, this is usually done by applying the induction hypothesis. For all rules whose premises require to typecheck some messages (*e.g.* POUT, PIFL,...), we apply Lemma 12 to show that these messages, once renamed, still have the correct type in the expanded environment. In the case of rule PLET, we apply Lemma 13 to show that the (renamed) destructors still have the correct type in the expanded environment.

As an example, let us detail the case of rule POUT. In that case, then $P = \mathtt{out}(M).P'$, $Q = \mathtt{out}(N).Q'$ for some M, N, P', Q', and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash P' \sim Q' \to C'} \qquad \dfrac{\Pi''}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}}{\Gamma \vdash P \sim Q \to C = C' \cup_\forall c}.$$

By Lemma 12, for all $\Gamma' \in \mathrm{branches}([\,\Gamma\,]_i^n)$, there exists a proof $\Pi'_{\Gamma'}$ of the type judgement $\Gamma' \vdash [\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma : \mathtt{LL} \to [\,c\,]_i^\Gamma$.

Moreover, by induction, there exists $C'' \subseteq [\,C'\,]_i^n$ and a proof Π'' of the type judgement $[\,\Gamma\,]_i^n \vdash [\,P'\,]_i^\Gamma \sim [\,Q'\,]_i^\Gamma \to C''$. We can show (again, with some technical properties omitted in this proof sketch) that this implies, for all branches Γ' of $[\,\Gamma\,]_i^n$, that $\Gamma' \vdash [\,P'\,]_i^\Gamma \sim [\,Q'\,]_i^\Gamma \to C_{\Gamma'}$ for some $C_{\Gamma'} \subseteq C'' \subseteq [\,C'\,]_i^n$.

In addition, we have $[\,P\,]_i^\Gamma = [\,\mathtt{out}(M).P'\,]_i^\Gamma = \mathtt{out}([\,M\,]_i^\Gamma).[\,P'\,]_i^\Gamma$, and similarly for process Q $[\,Q\,]_i^\Gamma = \mathtt{out}([\,N\,]_i^\Gamma).[\,Q'\,]_i^\Gamma$. Therefore, using $\Pi_{\Gamma'}$, $\Pi'_{\Gamma'}$ and rule POUT, we have for any branch $\Gamma' \in \mathrm{branches}([\,\Gamma\,]_i^n)$ that $\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_{\Gamma'} \cup_\forall [\,c\,]_i^\Gamma \subseteq [\,C'\,]_i^n \cup_\forall [\,c\,]_i^\Gamma$.

It can then be proved that typechecking processes in all branches of $[\,\Gamma\,]_i^n$ like this implies they typecheck in $[\,\Gamma\,]_i^n$ itself, *i.e.*

$$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C''' \subseteq [\,C'\,]_i^n \cup_\forall [\,c\,]_i^\Gamma = [\,C\,]_i^n$$

which proves the claim for this case. □

Using this lemma, we then show that any number of (renamed) copies of P, Q in parallel typecheck:

Lemma 15 (Typing n sessions)**.** *Consider processes P, Q such that $\Gamma \vdash P \sim Q \to C$ for some Γ, C. Then for all $n \in \mathbb{N}$ then for all $n \geq 1$, there exists $C' \subseteq \cup_{\times 1 \leq i \leq n} [\,C\,]_i^n$ such that*

$$[\,\Gamma\,]^n \vdash [\,P\,]_1^\Gamma \mid \ldots \mid [\,P\,]_n^\Gamma \sim [\,Q\,]_1^\Gamma \mid \ldots \mid [\,Q\,]_n^\Gamma \to C'$$

where $[\,\Gamma\,]^n$ is defined as $\bigcup_{1 \leq i \leq n} [\,\Gamma\,]_i^n$.

Proof sketch. Note that the union $\bigcup_{1 \leq i \leq n} [\,\Gamma\,]_i^n$ is well-defined, as for any indices $i \neq j$, we have $\mathrm{dom}([\,\Gamma\,]_i^n) \cap \mathrm{dom}([\,\Gamma\,]_j^n) \subseteq \mathcal{K} \cup \mathcal{N}$, and the types associated to keys and nonces are the same in each $[\,\Gamma\,]_i^n$.

The property follows from Lemma 14. Indeed, that lemma guarantees that for all $i \in [\![1, n]\!]$, $[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_i$ for some $C_i \subseteq [\,C\,]_i^n$. Using technical lemmas such as Lemma 1, we

can then show that $[\,\Gamma\,]^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_i'$ for some C_i'. More precisely, the environments in the elements of C_i do not contain all renamed keys, names and variables, but only those indexed with i, similarly to $[\,\Gamma\,]_i^n$. Thus, typechecking in $[\,\Gamma\,]^n$ adds all the other renamed elements to these environments: performing this operation on C_i yields constraint C_i'.

By applying rule PPar $n-1$ times, we thus have

$$[\,\Gamma\,]^n \vdash [\,P\,]_i^\Gamma 1 \mid \ldots \mid [\,P\,]_i^\Gamma n \sim [\,Q\,]_i^\Gamma 1 \mid \ldots [\,Q\,]_i^\Gamma n \to \cup_{\times 1\leq i\leq n} C_i'$$

Only a technical point remains to be proved: that $\cup_{\times 1\leq i\leq n} C_i' \subseteq \cup_{\times 1\leq i\leq n} [\,C\,]_i^n$. We omit this proof here, as it is more technical than insightful, and provide it in Appendix A.2 instead. □

Theorem 2 then follows from Lemma 15 and the soundness theorem (Theorem 1).

Indeed, consider P, Q and P', Q' that typecheck with respectively C and C' in Γ, and assume they satisfy the assumptions on finite and infinite types from Theorem 2. Using Lemma 15 and rule PPar we can show (we omit the details here) that for any n, P' in parallel with n (renamed) copies of P and Q' in parallel with n (renamed) copies of Q typecheck with (a subset of) constraint set $C' \cup_\times (\cup_{\times 1\leq i\leq n} [\,C\,]_i^n)$. These two processes are not replicated: Theorem 1 then applies, and if we know that $C' \cup_\times (\cup_{\times 1\leq i\leq n} [\,C\,]_i^n)$ is consistent for any n, implies that they are trace equivalent. That is, that P' in parallel with n copies of P and Q' in parallel with n copies of Q are trace equivalent for any n. Therefore, P' in parallel with $!P$ and Q' in parallel with $!Q$ are trace equivalent, which proves Theorem 2.

Theorem 1 requires to check consistency of one constraint set. Theorem 2 now requires to check consistency of an infinite family of contraint sets. Instead of *deciding* consistency, we provide in the next sections a procedure that checks a slightly stronger condition, which allows us to efficiently prove consistency of this infinite family by only considering a finite number of its elements.

2.7 Checking consistency

2.7.1 Procedure for consistency

As defined earlier (Definition 14), checking consistency of a set of constraints basically amounts to checking static equivalence of the corresponding frames, for any possible well-typed instantiations of their variables.

We devise a procedure `check_const`(C) for checking consistency of a constraint set C, depicted in Figure 2.9. Basically, we need to make sure that any equality test the attacker may perform on terms constructed from the left side of the constraint yields the same result when performed on the right side.

Using properties established in Section 2.5, we are able to show that it suffices to check that the messages in the constraints (without applying any attacker recipe) satisfy the same equalities.

The procedure then works as follows:

- First, variables of refined type $[\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!]$ are replaced by m on the left-hand-side of the constraint and n on the right-hand-side.
- Second, we check that terms have the same shape (encryption, signature, hash) on the left and on the right and that asymmetric encryption and hashes cannot be reconstructed by the adversary (that is, they contain some fresh, secret nonce).

- The most important step consists in checking that the terms on the left satisfy the same equalities than the ones on the right. Whenever two left terms M and N are unifiable, their corresponding right terms M' and N' should be equal after applying a similar instantiation.

From now on, we only consider constraint sets that can actually be generated when typing processes, as these are the only ones for which we need to check consistency.

Formally, the procedure `check_const` is described in Figure 2.9. It consists of three steps. First, we replace variables with refinements of finite nonce types by their left and right values. In particular a variable with a union type is not associated with a single value and thus cannot be replaced. This is why the branching operation needs to be performed when expanding environments containing refinements with types of the form $\tau_n^{l,\infty}$. Second, we check that the resulting constraints have the same shape. Finally, as soon as two constraints $M \sim M'$ and $N \sim N'$ are such that M, N are unifiable, we roughly check that $M' = N'$, and conversely. The actual condition, as seen in Figure 2.9, is slightly more involved, especially when the constraints contain variables of refined types with infinite nonce types. More precisely, the actual condition that would need to be checked is that whenever M, N can be unified with a unifier μ, then any substitution μ' such that μ and μ' are well-typed unifies M' and N' (and conversely). Indeed, the type system guarantees that the instantiations of variables that can occur in actual executions are well-typed (see Lemma 10). We consider for the procedure a stronger, but easier to ensure condition. We discard cases where M, N can only be unified by unifiers that do not respect the refinement types, as such instantiations cannot occur in actual executions. Moreover when M, N can be unified by μ, we check that M' and N' are equal, after replacing each variable with a refinement type (that μ respects) with the corresponding value, as well as instantiating each variable of type `LL` to which μ associates a nonce n with this same nonce n. This overapproximates the actual condition stated earlier: indeed, any well-typed μ' would respect these two conditions.

Example 5. *Consider the simplified model of Helios from Example 2. When typechecked with appropriate key types (as detailed in Introduction of this chapter), these processes yield constraint sets containing notably the following two constraints.*

$$\{\ \mathtt{aenc}(\langle 0, r_a\rangle, \mathtt{pk}(k_s)) \sim \mathtt{aenc}(\langle 1, r_a\rangle, \mathtt{pk}(k_s)),\\ \mathtt{aenc}(\langle 1, r_b\rangle, \mathtt{pk}(k_s)) \sim \mathtt{aenc}(\langle 0, r_b\rangle, \mathtt{pk}(k_s))\ \}$$

For simplicity, consider the set c containing only these two constraints, together with a typing environment Γ where r_a and r_b are respectively given types $\tau_{r_a}^{\mathtt{HH},1}$ and $\tau_{r_b}^{\mathtt{HH},1}$, and k_s is given type $\mathrm{key}^{\mathtt{HH}}(T)$ for some T.

The procedure `check_const`$(\{(c,\Gamma)\})$ *can detect that the constraint c is consistent and returns* `true`*. Indeed, as c does not contain variables,* $\mathtt{step1}_\Gamma(c)$ *simply returns (c,Γ).* $\mathtt{step2}_\Gamma(c)$ *then returns* `true`*, since the messages appearing in c are messages asymmetrically encrypted with secret keys, which contain a secret nonce (r_a or r_b) directly under pairs. Finally* $\mathtt{step3}_\Gamma(c)$ *trivially returns* `true`*, as the messages* $\mathtt{aenc}(\langle 0, r_a\rangle, \mathtt{pk}(k_s))$ *and* $\mathtt{aenc}(\langle 1, r_b\rangle, \mathtt{pk}(k_s))$ *cannot be unified, as well as the messages* $\mathtt{aenc}(\langle 1, r_a\rangle, \mathtt{pk}(k_s))$ *and* $\mathtt{aenc}(\langle 0, r_b\rangle, \mathtt{pk}(k_s))$*.*

Consider now the following set c', where encryption has not been randomised:

$$c' = \{\ \mathtt{aenc}(0, \mathtt{pk}(k_s)) \sim \mathtt{aenc}(1, \mathtt{pk}(k_s)),\\ \mathtt{aenc}(1, \mathtt{pk}(k_s)) \sim \mathtt{aenc}(0, \mathtt{pk}(k_s))\ \}$$

The procedure `check_const`$(\{(c',\Gamma)\})$ *returns* `false`*. Indeed, contrary to the case of c,* $\mathtt{step3}_\Gamma(c')$ *fails, as the encrypted message do not contain a secret nonce. Actually, the corresponding frames are indeed not statically equivalent since the adversary can reconstruct the encryption of* 0 *and* 1 *with the key* $\mathtt{pk}(k_s)$ *(in his initial knowledge), and check for equality.*

$\texttt{step1}_\Gamma(c) := (\llbracket c \rrbracket_{\sigma_F, \sigma'_F}, \Gamma')$, with

$$F := \{x \in \mathrm{dom}(\Gamma) \mid \exists m, n, l, l'.\ \Gamma(x) = \llbracket \tau_m^{l,1}\ ;\ \tau_n^{l',1} \rrbracket\}$$

and σ_F, σ'_F defined by

$$\begin{cases} \bullet\ \mathrm{dom}(\sigma_F) = \mathrm{dom}(\sigma'_F) = F \\ \bullet\ \forall x \in F.\ \forall m, n, l, l'.\ \Gamma(x) = \llbracket \tau_m^{l,1}\ ;\ \tau_n^{l',1} \rrbracket \Rightarrow \sigma_F(x) = m\ \wedge\ \sigma'_F(x) = n \end{cases}$$

and Γ' is $\Gamma|_{\mathrm{dom}(\Gamma)\setminus F}$ extended with $\Gamma'(n) = \tau_n^{l,1}$ for all nonce n such that $\tau_n^{l,1}$ occurs in Γ.

$\texttt{step2}_\Gamma(c)$:= check that for all $M \sim N \in c$, M and N are both

- $\texttt{enc}(M', M'')$, $\texttt{enc}(N', N'')$ where M'', N'' are either
  - keys k, k' where $\exists T.\ \Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - or a variable x such that $\exists T.\ \Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
- or encryptions $\texttt{aenc}(M', M'')$, $\texttt{aenc}(N', N'')$ where
  - M' and N' contain directly under pairs a nonce n such that $\Gamma(n) = \tau_n^{\mathtt{HH},a}$ or a secret key k such that $\exists T, k'.\ \Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$ or $\Gamma(k', k) <: \mathrm{key}^{\mathtt{HH}}(T)$, or a variable x such that $\exists m, n, a.\ \Gamma(x) = \llbracket \tau_m^{\mathtt{HH},a}\ ;\ \tau_n^{\mathtt{HH},a} \rrbracket$, or a variable x such that $\exists T.\ \Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - M'' and N'' are either
    - public keys $\texttt{pk}(k)$, $\texttt{pk}(k')$ where $\exists T.\ \Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
    - or public keys $\texttt{pk}(x)$, $\texttt{pk}(x)$ where $\exists T.\ \Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
    - or a variable x such that $\exists T, T'.\ \Gamma(x) = \mathrm{pkey}(T)$ and $T <: \mathrm{key}^{\mathtt{HH}}(T')$;
- or hashes $\texttt{h}(M')$, $\texttt{h}(N')$, where M', N' similarly contain a secret value under pairs;
- or signatures $\texttt{sign}(M', M'')$, $\texttt{sign}(N', M'')$ where M'', N'' are either
  - keys k, k' where $\exists T.\ \Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
  - or a variable x such that $\exists T.\ \Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;

$\texttt{step3}_\Gamma(c)$:= If for all $M \sim M'$ and $N \sim N' \in c$ such that M, N are unifiable with a most general unifier μ, and such that

$$\forall x \in \mathrm{dom}(\mu).\ \exists l, l', m, p.\ (\Gamma(x) = \llbracket \tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty} \rrbracket) \Rightarrow (x\mu \in \mathcal{X}\ \vee\ \exists i.\ x\mu = m_i)$$

we have

$$M'\alpha\theta = N'\alpha\theta$$

where

$$\forall x \in \mathrm{dom}(\mu).\ \forall l, l', m, p, i.\ (\Gamma(x) = \llbracket \tau_m^{l,\infty}\ ;\ \tau_p^{l',\infty} \rrbracket\ \wedge\ \mu(x) = m_i) \Rightarrow \theta(x) = p_i$$

and α is the restriction of μ to $\{x \in \mathrm{dom}(\mu) \mid \Gamma(x) = \mathtt{LL}\ \wedge\ \mu(x) \in \mathcal{N}\}$;
and if the symmetric condition for the case where M', N' are unifiable holds as well, then return `true`.

$\texttt{check\_const}(C)$:= for all $(c, \Gamma) \in C$, let $(c_1, \Gamma_1) := \texttt{step1}_\Gamma(c)$ and check that $\texttt{step2}_{\Gamma_1}(c_1) = \texttt{true}$ and $\texttt{step3}_{\Gamma_1}(c_1) = \texttt{true}$.

Figure 2.9 – Procedure for checking consistency.

2.7.2 Soundness in the bounded case

For constraint sets without infinite nonce types, `check_const` entails consistency. We now sketch the proof for this claim. The reader may notice that the procedure does consider the case of infinite nonce types: this will be useful to prove consistency of replicated constraints in the next section. As before, we only sketch the structure of the proof here. All details are provided in Appendix A.3.

Formally, we prove the following.

Theorem 3. *Let C be a set of constraints such that*

$$\forall (c, \Gamma) \in C.\, \forall l, l', m, p.\, \Gamma(x) \neq [\![\tau_m^{l,\infty}\, ;\, \tau_p^{l',\infty}]\!].$$

If `check_const`$(C) =$ `true`*, then C is consistent.*

We prove this theorem by first showing that the first step of the procedure preserves consistency.

Lemma 16. *Consider a constraint (c, Γ) obtained by typing some processes. Let $(\overline{c}, \overline{\Gamma}) =$ `step1`$_\Gamma(c)$. If $\overline{c}$ is consistent in $\overline{\Gamma}$, then c is consistent in Γ.*

Proof sketch. Assume $\overline{c}$ is consistent in $\overline{\Gamma}$. Let $c' \subseteq c$ and $\Gamma' \subseteq \Gamma$, such that $\mathrm{dom}(\Gamma')$ contains all bikeys and names from $\mathrm{dom}(\Gamma)$, as well as all variables occurring in c'. Let σ, σ' be two substitutions such that ${\Gamma'}_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : {\Gamma'}_{\mathcal{X}} \to c_\sigma$ (for some $c_\sigma \subseteq [\![c']\!]_{\sigma,\sigma'}$).

To prove the claim, we need to show that the frames $\phi_{\mathrm{LL}}^{\Gamma} \cup \phi_\ell([\![c']\!]_{\sigma,\sigma'}))$ and $(\phi_{\mathrm{LL}}^{\Gamma} \cup \phi_r([\![c']\!]_{\sigma,\sigma'}))$ are statically equivalent. The idea of the proof is to consider the constraint set c'' obtained by instantiating all variables with refinement types in c' with the corresponding values. Since σ, σ' are well-typed, they correctly instantiate such variables with the same values. Hence, $[\![c']\!]_{\sigma,\sigma'}$ can be seen as the instantiation of c'' with some $\overline{\sigma}$, $\overline{\sigma'}$ that are restrictions of σ and σ' to variables without refinement types.

On the other hand, c'' is a subset of $\overline{c}$, by construction, and we can show (we omit the technical details here) that $\overline{\sigma}$, $\overline{\sigma'}$, being restrictions of σ, σ' are well-typed in a subset of $\overline{\Gamma}$ that satisfies the conditions from the definition of consistency. From this, we get that $\phi_{\mathrm{LL}}^{\Gamma} \cup \phi_\ell([\![c'']\!]_{\overline{\sigma},\overline{\sigma'}}))$ and $(\phi_{\mathrm{LL}}^{\Gamma} \cup \phi_r([\![c'']\!]_{\overline{\sigma},\overline{\sigma'}}))$ are statically equivalent, which proves the claim. □

It then suffices to check the consistency of the constraint and environment produced by `step1`$_\Gamma(c)$. Provided that `step2`$_\Gamma$ holds, we show that this constraint is saturated in the sense that any message obtained by the attacker by decomposing terms in the constraint already occurs in the constraint.

Lemma 17. *Consider a constraint (c, Γ) obtained by typing some processes. Let $(\overline{c}, \overline{\Gamma}) =$ `step1`$_\Gamma(c)$, and assume* `step2`$_{\overline{\Gamma}}(\overline{c})$ *holds. Consider ground, well-typed instantiations σ, σ',* i.e. *$\overline{\Gamma}_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \overline{\Gamma}_{\mathcal{X}} \to c_\sigma$ for some $c_\sigma \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}$. Also consider an attacker recipe R such that* $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell(\overline{c}))$. *Assume $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow \neq \bot$ and $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow \neq \bot$.*

Then there exists a recipe R' without destructors, i.e. *in which* `dec`*,* `adec`*,* `checksign`*, π_1, π_2, do not appear, such that*

- $\mathrm{vars}(R') \subseteq \mathrm{vars}(R)$,

- $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$,
- $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))$.

In addition, if

Proof sketch. This property is proved by induction on the attacker recipe R. Basically, when R starts with a constructor, the claim follows easily from the induction hypothesis. The cases of destructors are more involved. The idea is that `step2` guarantees that the messages in the constraint cannot be opened by the attacker. Hence, the destructor in R is actually being applied to a message constructed by the attacker, and by removing both the constructor and the destructor, we obtain a smaller recipe that produces the same message, and to which the induction hypothesis can be applied.

We present here the case where $R = \mathtt{dec}(S, K)$ for some recipes S, K. Since R successfully evaluates when applied to either side of $\overline{c}$, K must evaluate to a key when applied to it. In addition we can show (again, we omit technical details here) that the messages in a constraint obtained by typing (and thus those in $\overline{c}$) have type `LL` with a subset of this constraint. By Lemma 7, K applied on either side of $\overline{c}$ is therefore a key of type `LL`.

In addition, by induction, we may replace S with a recipe without destructors S' that produces the same messages when applied to the constraint. Since R successfully evaluates, this means S' applied to either side of the constraint evaluates to an encryption with k. As k is of type `LL`, by `step2`, this encryption cannot already be present in the constraint. Thus it was constructed by the attacker: $S' = \mathtt{enc}(S'', K')$ for some S'', K'' without destructors (and such that K'' produces the correct key). When applied to the constraint, R thus produces the same result as S'': using this S'' as the R' specified in the statement of the lemma proves the claim. □

In addition, if follows rather easily that if the checks in `step2` succeed, then the constraint only contains messages which cannot be reconstructed by the attacker from the rest of the constraint. Indeed, they are all encryptions or signatures with secret keys, or contain a secret nonce. Using Lemma 17 and this property, we finally prove that the simple unification tests performed in `step3` are sufficient to ensure static equivalence of each side of the constraint for any well-typed instantiation of the variables.

Lemma 18. *Consider a constraint (c, Γ) obtained by typing some processes. Let $(\overline{c}, \overline{\Gamma}) = \mathtt{step1}_\Gamma(c)$, and assume $\mathtt{step2}_{\overline{\Gamma}}(\overline{c})$ and $\mathtt{step3}_{\overline{\Gamma}}(\overline{c})$ hold. Consider ground, well-typed instantiations σ, σ',* i.e. *$\overline{\Gamma}_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \overline{\Gamma}_{\mathcal{X}} \to c_\sigma$ for some $c_\sigma \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}$. Also consider attacker recipes R, S such that* $\mathrm{vars}(R) \cup \mathrm{vars}(S) \subseteq \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell(\overline{c}))$.

Then

$$(R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})))\downarrow = (S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})))\downarrow \iff (R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})))\downarrow = (S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})))\downarrow .$$

Proof sketch. We only give the main ideas of this proof, and omit here all technical details. Basically, we first use Lemma 7 to take care of the case where either of the recipes evaluate to $\bot$. Then, assuming all recipes successfully evaluate, Lemma 17 allows us to consider only recipes R, S without destructors.

We then do the proof by induction on (the sum of) the sizes of R, S. Both directions of the equivalence are similar, we show ($\Rightarrow$). Assume $(R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})))\downarrow = (S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})))\downarrow$. Since R, S do not contain destructors, and successfully evaluate, this means that $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$. Therefore, we can distinguish three cases regarding R, S:

- either none of them are variables: then they necessarily start with the same symbol, which can be a name or a constructor. In the first case, the claim is trivial, and in the second case it follows by induction: indeed, if $R = f(R')$ and $S = f(S')$, we can apply the induction hypothesis to R', S' to get that they produce the same message when applied on the right. It then follows that the same holds for R, S and the claim holds.
- or one of them is a variable and the other is not: this case is impossible by the observation that no message in the constraints can be reconstructed from the others.
- or both of them are variables: then the test being performed is simply an equality check between messages of the instantiated constraint. Using the fact that $\mathtt{step3}_{\overline{\Gamma}}(\overline{c})$ holds, we can prove that this check also succeeds on the right side of the instantiated constraint. □

Lemma 18 proves that, if the procedure succeeds on c, Γ, then $\overline{c}$ is consistent in $\overline{\Gamma}$, where $(\overline{c}, \overline{\Gamma}) = \mathtt{step1}_{\Gamma}(c)$. By Lemma 16, this proves that c is then consistent in Γ, which proves the soundness result (Theorem 3).

As a direct consequence of Theorems 1 and 3, we now have a procedure to prove trace equivalence of processes without replication.

For proving trace equivalence of processes with replication, we need to check consistency of an infinite family of constraint sets, as prescribed by Theorem 2. As mentioned earlier, not only the number of constraints is unbounded, but the size of the type of some (replicated) variables is also unbounded (*i.e.* of the form $\bigvee_{j=1}^{n} [\![\tau_{m_j}^{l,1} \,;\, \tau_{p_j}^{l',1}]\!]$). We use here two ingredients: we first show that it is sufficient to apply our procedure to two constraints only. Second, we show that our procedure applied to variables with replicated types, *i.e.* nonce types of the form $\tau_n^{l,\infty}$ implies consistency of the corresponding constraints with types of unbounded size.

2.7.3 Unbounded case: two constraints suffice

Consistency of a constraint set C does not guarantee consistency of $\cup_{\times 1 \le i \le n} [\, C \,]_i^n$. For example, consider

$$C = \{(\{\mathtt{h}(m) \sim \mathtt{h}(p)\}, [m : \tau_m^{\mathtt{HH},\infty}, p : \tau_p^{\mathtt{HH},1}])\}$$

which can be obtained when typing

$$\begin{array}{l}\mathtt{new}\ m : \tau_m^{\mathtt{HH},\infty}.\ \mathtt{new}\ p : \tau_p^{\mathtt{HH},1}.\ \mathtt{out}(\mathtt{h}(m)) \sim \\ \mathtt{new}\ m : \tau_m^{\mathtt{HH},\infty}.\ \mathtt{new}\ p : \tau_p^{\mathtt{HH},1}.\ \mathtt{out}(\mathtt{h}(p)).\end{array}$$

C is consistent: since m, p are secret, the attacker cannot distinguish between their hashes. However $\cup_{\times 1 \le i \le n} [\, C \,]_i^n$ contains (together with some environment):

$$\{\mathtt{h}(m_1) \sim \mathtt{h}(p), \mathtt{h}(m_2) \sim \mathtt{h}(p), \ldots, \mathtt{h}(m_n) \sim \mathtt{h}(p)\}$$

which is not, since the attacker can notice that the value on the right is always the same, while the value on the left is not.

Note however that the inconsistency of $\cup_{\times 1 \le i \le n} [\, C \,]_i^n$ would have been discovered when checking the consistency of two copies of the constraint set only. Indeed, $[\, C \,]_1^n \cup_{\times} [\, C \,]_2^n$ contains (together with some environment):

$$\{\mathtt{h}(m_1) \sim \mathtt{h}(p), \mathtt{h}(m_2) \sim \mathtt{h}(p)\}$$

which is already inconsistent, for the same reason.

Actually, checking consistency (with our procedure) of two constraints $[\,C\,]_1^n$ and $[\,C\,]_2^n$ entails consistency of $\cup_{\times 1\leq i\leq n}[\,C\,]_i^n$. Note that this does not mean that consistency of $[\,C\,]_1^n$ and $[\,C\,]_2^n$ implies consistency of $\cup_{\times 1\leq i\leq n}[\,C\,]_i^n$. Instead, our procedure ensures a stronger property, for which two constraints suffice.

Theorem 4. *Let C and C' be two constraint sets that*

- *do not share any common variable,* i.e.

$$\forall (c,\Gamma)\in C.\ \forall (c',\Gamma')\in C'.\ \mathrm{dom}(\Gamma_{\mathcal{X}})\cap \mathrm{dom}(\Gamma'_{\mathcal{X}})=\emptyset;$$

- *only share nonces which have the same finite nonce type,* i.e.

$$\forall (c,\Gamma)\in C.\ \forall (c',\Gamma')\in C'.\ \forall m\in \mathrm{dom}(\Gamma)\cap\mathrm{dom}(\Gamma')\cap\mathcal{N}.\ \exists l.\ \Gamma(m)=\Gamma'(m)=\tau_m^{l,1};$$

- *only share keys which are paired in the same way and have the same types,* i.e.

$$\begin{array}{l}\forall (c,\Gamma)\in C.\ \forall (c',\Gamma')\in C'.\ \forall k\in \mathrm{keys}(\Gamma)\cap\mathrm{keys}(\Gamma').\forall k'\in\mathcal{K}.\\ \quad((k,k')\in\mathrm{dom}(\Gamma)\Leftrightarrow(k,k')\in\mathrm{dom}(\Gamma'))\ \wedge\ ((k',k)\in\mathrm{dom}(\Gamma)\Leftrightarrow(k',k)\in\mathrm{dom}(\Gamma'))\end{array}$$

  and

$$\forall (c,\Gamma)\in C.\ \forall (c',\Gamma')\in C'.\ \forall (k,k')\in\mathrm{dom}(\Gamma)\cap\mathrm{dom}(\Gamma').\ \Gamma(k,k')=\Gamma'(k,k').$$

For all $n\in\mathbb{N}$, if $\texttt{check\_const}([\,C\,]_1^n\cup_\times[\,C\,]_2^n\cup_\times[\,C'\,]_1^n)=\texttt{true}$, *then*

$$\texttt{check\_const}(\cup_{\times 1\leq i\leq n}[\,C\,]_i^n)\cup_\times[\,C'\,]_1^n)=\texttt{true}.$$

Proof sketch. We only sketch the ideas of the proof here, all technical details can be found in Appendix A.4. To prove Theorem 4, we first (easily) show that if

$$\texttt{check\_const}([\,C\,]_1^n\cup_\times[\,C\,]_2^n\cup_\times[\,C'\,]_1^n)=\texttt{true},$$

then the first two steps of the procedure `check_const` can be successfully applied to each element of $(\cup_{\times 1\leq i\leq n}[\,C\,]_i^n)\cup_\times[\,C'\,]_1^n$.

However the case of the third step is more intricate. When applying the procedure `check_const` to an element of the constraint set $(\cup_{\times 1\leq i\leq n}[\,C\,]_i^n)\cup_\times[\,C'\,]_1^n$, if `step3` fails, then the constraint contains an inconsistency, *i.e.* elements $M\sim M'$ and $N\sim N'$ for which the unification condition from `step3` does not hold. More precisely, M and N can be unified with an acceptable (in the sense of `step3`) unifier, while M' and N' are still different once their variables with refinement and `LL` types are instantiated as in `step3`.

Then we show that we can find a similar inconsistency when considering only the first two constraint sets, *i.e.* in $[\,C\,]_1^n\cup_\times[\,C\,]_2^n\cup_\times[\,C'\,]_1^n$. This is done by reindexing the nonces and variables. The proof actually requires a careful examination of the structure of the constraint set $(\cup_{\times 1\leq i\leq n}[\,C\,]_i^n)\cup_\times[\,C'\,]_1^n$, to establish this reindexing. More precisely, we examine the origin of the elements $M\sim M'$ and $N\sim N'$ in the constraint set $(\cup_{\times 1\leq i\leq n}[\,C\,]_i^n)\cup_\times[\,C'\,]_1^n$.

- If they both come from the same constraint set $[\,C\,]_i^n$ for some i: then we can show that the same inconsistency occurs in $[\,C\,]_1^n$, which is just a renaming of $[\,C\,]_i^n$.

- If they come from two sets $[\,C\,]_i^n$ and $[\,C\,]_j^n$, for some $i \neq j$: then similarly the same inconsistency occurs with the corresponding elements of $[\,C\,]_1^n$ and $[\,C\,]_2^n$. Note that we still need to consider two different indices (i and j, or 1 and 2) to retrieve the inconsistency. Indeed, it might be for instance that this inconsistency uses two renamings of the same element of the original C, and is thus not present in $[\,C\,]_1^n$ alone.
- Finally if one of them comes from some $[\,C\,]_i^n$ and the other from $[\,C'\,]_1^n$: then similarly the same inconsistency can be found between $[\,C\,]_1^n$ and $[\,C\,]_2^n$.

□

Note that the rather lengthy hypotheses for this theorem only serve to ensure that the environments in all the constraint sets are compatible, but do not really change the intuition of the result. They are clearly satisfied when considering constraints produced by typechecking processes that satisfy the assumptions of Theorem 2.

We now know that checking two copies of the constraint set suffices to establish consistency of any number of copies. However, in these two copies, the infinite nonce types are still unfolded into union types of size n. The next step is to reduce the problem to checking constraints with fixed, bounded types.

2.7.4 Reducing the size of types

We prove that the procedure `check_const` applied to replicated types, *i.e.* nonce and refinement types with $a = \infty$, implies consistency of corresponding constraints with unbounded types.

Theorem 5. *Let C, and C' be two constraint sets without any common variable (with the same assumptions as in Theorem 4). Then:*

$$\texttt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \texttt{true} \;\Rightarrow\; \forall n.\ \texttt{check\_const}([\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n) = \texttt{true}.$$

Proof sketch. Again here, it is rather easy to show that if $\texttt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \texttt{true}$ then the first two steps of the procedure `check_const` can successfully be applied to each element of $[\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n$. The case of `step3` is more involved. The property holds thanks to the condition on the most general unifier expressed in `step3`. Intuitively, this condition is written in such a way that if, when applying `step3` to sets of constraints in $[\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n$, two messages can be unified, then the corresponding messages (with infinite, non expanded types) in $[\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1$ can be unified with a most general unifier that also satisfies the condition. The proof uses this idea to show that if `step3` succeeds on all sets in $[\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1$, then it also succeeds on the sets in $[\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n$. The detailed proof is provided in Appendix A.4. □

2.7.5 Checking the consistency of the infinite constraint

Theorems 3, 4, and 5 prove that our procedure `check_const` is sound for checking consistency of replicated constraints.

Theorem 6. *Let C, and C' be two constraint sets without any common variable (in the sense of Theorem 4). Then*

$$\texttt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \texttt{true} \;\Rightarrow\; \forall n.\ [\,C'\,]_1^n \cup_\times (\cup_{\times 1 \leq i \leq n} [\,C\,]_i^n) \text{ is consistent.}$$

This theorem is a direct consequence of Theorems 3, 4, and 5.

It shows that it suffices to check to renamed copies of the constraints, without having to extend the types they contain, to ensure consistency of the unbounded constraints produced when typechecking replicated processes.

2.8 Conclusion

Summarising all the results presented in this chapter, we have introduced a type system that typechecks two processes, and produces constraints, as well as a procedure to check the consistency of this constraint. We have proved that the type system, together with the procedure, are sound, in the sense that they imply trace equivalence, both for processes with bounded and unbounded numbers of sessions. Formally, the following theorem summarises Theorems 2, and 6:

Theorem 7. *Consider processes P, Q, P' ,Q' such that P, Q and P', Q' do not share any variable. Consider Γ, containing only keys and nonces with finite key or nonce types. Assume that P and Q only bind nonces and keys with infinite nonce types,* i.e. *using* $\texttt{new}\ m : \tau_m^{l,\infty}$ *and* $\texttt{new}\ k : \text{seskey}^{l,\infty}(T)$ *for some label l and type T; while P' and Q' only bind nonces and keys with finite types,* i.e. *using* $\texttt{new}\ m : \tau_m^{l,1}$ *and* $\texttt{new}\ k : \text{seskey}^{l,1}(T)$.

Let us abbreviate by $\texttt{new}\ \overline{n}$ *the sequence of declarations of each nonce $m \in \text{dom}(\Gamma)$ and session key k such that $\Gamma(k,k) = \text{seskey}^{l,1}(T)$ for some l, T.*

If

- $\Gamma \vdash P \sim Q \to C$,
- $\Gamma \vdash P' \sim Q' \to C'$,
- $\texttt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \texttt{true}$,

then

$$\texttt{new}\ \overline{n}.\ ((!P) \mid P') \approx_t \texttt{new}\ \overline{n}.\ ((!Q) \mid Q').$$

This constitutes an efficient way of checking trace equivalence of processes with or without replication, and even processes with a replicated and an unreplicated part.

In the next chapter, we give relevant examples where this feature is useful. In addition, we designed a prototype implementation of a typechecker for the type system, and of the procedure for consistency, to experimentally validate the efficiency of our approach.

Chapter 3

Examples and Experimental Results

3.1 Introduction

In the previous chapter, we introduced our type system for equivalence, together with the companion procedure for consistency. We will now illustrate how this type system can be used to typecheck protocols. We first study the Helios voting protocol already evoked in Section 2.1. We describe how our typesystem can be used to show that it guarantees vote secrecy, assuming the ballot box is not trusted. This case study illustrates well an interesting feature of our type system: we can consider processes involving a mix of some replicated and unreplicated parts. This is notably useful when modelling voting systems, to represent scenarios that honest voters are not allowed to revote (hence their processes are not replicated) but we still want to prove security assuming the attacker controls any number of dishonest voters (represented by replicated processes).

As a second example, we study the Private Authentication protocol already mentioned in Section 2.2. In this protocol, an agent A is trying to authenticate to another agent B, who can accept or reject this request. The protocol aims to prevent an attacker from learning whether B accepted A's request or not. Studying this property illustrates another feature of our type system: its ability to prove equivalence even in case the execution is not uniform, *i.e.* branches differently on the left and on the right. This models the fact that, in the equivalence we prove for Private Authentication, authentication typically succeeds on the left and fails on the right, leading to non uniform execution.

Finally, to validate experimentally the efficiency of our approach, we implemented together with Niklas Grimm a prototype of a typechecker for our type system, as well as the procedure for consistency of the constraints. We apply it to several examples of protocols, notably our two case studies, as well as several key exchange protocols from the literature, for which we prove a strong secrecy property on the exchanged key. We then compare the performance of our prototype to that of other existing automated tools for symbolic process equivalence. The source code for the implementation, as well as the files from the benchmark, are available at [1].

3.2 Helios

In this section, we apply our type system to the Helios voting protocol described in Section 2.2. We show how our type system can prove vote privacy for Helios, and present the most interesting parts of the type derivation.

3.2.1 Modelling vote privacy for Helios

Recall that the protocol was informally described in Section 2.2 as follows

$$\begin{array}{ll} S \to V_i : & r_i \\ V_i \to S : & \mathtt{sign}(\mathtt{aenc}(\langle v_i, \langle r_i, r_i' \rangle \rangle, \mathtt{pk}(k_s)), k_i) \\ S \to V_1, \ldots, V_n : & v_1, \ldots, v_n \end{array}$$

Voters receive a random value r_i from the server (in clear and unauthenticated). They encrypt their vote with the election key $\mathtt{pk}(k_s)$, randomising the encryption with both r_i and a fresh value r_i' they generate. They sign this ciphertext with their private key and send it to the server (modelling an authenticated communication channel with the server). The server receives all ballots, and removes duplicate ciphertexts (we call this operation *weeding*). This phase is crucial, as otherwise the protocol is subject to an attack, as explained in introduction and Section 2.2. Finally, the server, who knows the decryption key k_s, decrypts the ballots, and publishes (in an arbitrary order) the list of votes in clear.

That model is an abstraction of the actual Helios protocol: in the real scheme, the voting server that collects the ballots and the tallying authority are distinct entities, and the decryption key k_s is split among several trustees that form the tallying authority. We consider here the case where the talliers are honest, as well as the server (although the attacker, who controls the network has the ability to block honest ballots, or send fake ballots to the server). Hence, we use a very abstract model for the tally, that conflates the tallying authority and the voting server into a single entity who collects ballots, decrypts them and computes the result.

In addition, we model the initial phase where the election key $\mathtt{pk}(k_s)$ is distributed to the voters before the actual election starts. We represent this distribution by having the server generate a fresh election key, and send it to voters in an authenticated way: the key is signed with some long-term key k_0, and we assume voters already know the verification key $\mathtt{vk}(k_0)$.

We model the role of a voter with private key k voting for candidate v by the following process $V(k, v)$:

$$\begin{array}{lll} V(k,v) & = & \mathtt{in}(x). \\ & & \mathtt{let}\ x_k = \mathtt{checksign}(x, \mathtt{vk}(k_0))\ \mathtt{in} \qquad \text{receiving the election public key} \\ & & \quad \mathtt{in}(x_r).\mathtt{new}\ r'. \\ & & \mathtt{out}(\mathtt{sign}(\mathtt{aenc}(\langle v, \langle x_r, r' \rangle \rangle, x_k), k)) \qquad \text{sending the signed ballot} \end{array}$$

The model of the role of the voting server is a bit more intricate. As already explained in earlier chapters (*e.g.* Section 2.1), we will model the privacy property by considering two honest voters Alice and Bob with secret keys k_1 and k_2, in parallel with other, compromised, voters. We will additionally call r_1' the nonce generated by Alice, and r_2' the one from Bob. In order to be able to consider an unbounded number of compromised voters, we split the voting server into two parts: one that handles only the ballots from Alice and Bob, and that will not be replicated, and one that handles all ballots from dishonest voters, that will be replicated. In order to prevent false attacks, we require that the server only proceeds to the tallying if ballots signed by Alice and Bob have correctly been received: otherwise, the attacker could simply prevent Alice from voting, and the result would only contain Bob's vote, which trivially breaks privacy.

Then, to be able to correctly perform weeding, the voting server dedicated to dishonest voters needs to know Alice and Bob's ciphertext. Actually, for technical reasons which will become clear later on, we model the weeding as comparing the random values used to randomise the

encryption, rather that the ciphertexts themselves. While this may not be possible or make sense in practice with the actual cryptographic primitive used, in our symbolic model, since the values of the random nonces generated by the voters are not published, comparing the random value or the whole ciphertext amounts to the same. Hence, we require that the voting server for Alice and Bob first receives their ballots, and sends privately to the other part of the voting server the two random values r'_1, r'_2 they contain. When receiving a dishonest ballot, the second part of the voting server will then compare its random nonce to r'_1, r'_2, and discard it if it is equal to one of the two. This private communication is modelled as sending these values symmetrically encrypted using a key k' that only the two parts of the voting server know. We take advantage of this step to also sends the secret key k of the election, generated by the server for Alice and Bob, to the other part of the server. This communication between the two parts of the server does not represent any real part of the protocol. It is simply a modelling artefact we need to use to represent the two parts of the server sharing their internal state, which is not natively possible in our pi-calculus.

Another point in our model is that we do not generate honest signature keys for dishonest voters. Instead, we consider that these are directly generated by the attacker, and the server for dishonest voters simply asks the attacker for a verification key to be used. For better readability, we omit this from the presentation, and simply assume here that dishonest votes are not signed. Finally, we model the publication of the result in a randomised order by simply outputting the votes in clear in a non-deterministic order, *i.e.* in two outputs in parallel.

The voting server for voters Alice and Bob with verification keys vk_1, vk_2 is modelled by the following process:

$S(vk_1, vk_2)$	=	$\texttt{new}\ k.\texttt{out}(\texttt{sign}(\texttt{pk}(k), k_0)).$	generating and sending the election key
		$\texttt{new}\ r_1.\texttt{new}\ r_2.\texttt{out}(r_1).\texttt{out}(r_2).$	sending the random values to A and B
		$\texttt{in}(x_1).\ \texttt{in}(x_2).$	receiving ballots
		$\texttt{let}\ y_1 = \texttt{checksign}(x_1, vk_1)\ \texttt{in}$	opening A's ballot
		$\quad \texttt{let}\ (z_{v1}, z_{r1}, z'_{r1}) = \texttt{adec}(y_1, k)\ \texttt{in}$	
		$\texttt{let}\ y_2 = \texttt{checksign}(x_2, vk_2)\ \texttt{in}$	opening B's ballot
		$\quad \texttt{let}\ (z_{v2}, z_{r2}, z'_{r2}) = \texttt{adec}(y_2, k)\ \texttt{in}$	
		$\texttt{out}(\texttt{enc}(\langle\langle z'_{r1}, z'_{r2}\rangle, k\rangle, k'))$	sending values to other server
		$(\texttt{out}(z_{v1}) \mid \texttt{out}(z_{v2}))$	publishing the result

Note that when opening a ballot, we wrote the decomposition of the plaintext into three variables $(z_{v1}, z_{r1}, z'_{r1})$ as a single destructor application for clarity, while in the actual process it consists in a succession of several destructors.

The voting server handling dishonest voters is then modelled by the following process:

S'	=	$\texttt{in}(y).\ \texttt{let}\ (y_{r1}, y_{r2}, y_k) = \texttt{dec}(y, k')\ \texttt{in}.$	receiving the values from the server
		$\texttt{in}(y_3).$	receiving a dishonest ballot
		$\texttt{let}\ (z_{v3}, z_{r3}, z'_{r3}) = \texttt{adec}(y_3, k)\ \texttt{in}$	opening the ballot
		$\texttt{if}\ z'_{r3} = y_{r1}\ \texttt{then}\ 0$	weeding
		$\texttt{else if}\ z'_{r3} = y_{r2}\ \texttt{then}\ 0$	
		$\quad \texttt{else}\ \texttt{out}(z_{v3})$	publish the vote in clear

There is no need for a process modelling dishonest voters: they are entirely controlled by the attacker. Finally we add another process *Setup*, in charge of publishing the public keys to the attacker:

$$Setup = \mathtt{out}(\mathtt{vk}(k_1), \mathtt{vk}(k_2), \mathtt{vk}(k_0)).$$

Altogether, our model for Helios, with Alice voting for v_1 and Bob for v_2 is then the following process:

$$Helios(v_1, v_2) = Setup \mid V(k_1, v_1) \mid V(k_2, v_2) \mid S(\mathtt{vk}(k_1), \mathtt{vk}(k_2)) \mid !S'$$

As explained earlier (introduction, Section 2.2), the vote privacy property states that the attacker should not be able to distinguish between Alice voting for 0 and Bob for 1 from Alice voting for 1 and Bob for 0[6]. Formally, we thus prove the following equivalence:

$$Helios(0, 1) \approx_t Helios(1, 0)$$

To help the proof go through, we actually slightly rewrite the process on the right into $Helios'(1, 0)$, which is identical to $Helios(1, 0)$, except that in the server for Alice and Bob, the final outputs of the votes in parallel are swapped, *i.e.* $\mathtt{out}(z_{v1}) \mid \mathtt{out}(z_{v2})$ is replaced with $\mathtt{out}(z_{v2}) \mid \mathtt{out}(z_{v1})$. This process clearly has the same behaviours as the original one, as the parallel branching is executed in a non deterministic order. However it makes the proof easier: in practice z_{v1} will contain 0 on the left and 1 on the right, while z_{v1} will contain 1 on the left and 0 on the right. Hence this swapping makes it so that the first message (z_{v1} on the left, z_{v2} on the right) always contains 0, and the second one 1.

3.2.2 Typechecking Helios

A key point when proving this equivalence is that we need to keep track precisely of the value of the contents of the two variables z_{v1} and z_{v2}, to be able to typecheck the final output in the server process. The second key point is the typechecking of the weeding phase. This phase is intended to prevent the attacker from resubmitting a copy of Alice's or Bob's ballot. We need precise enough types to express that when the weeding succeeds, the ballot we are decrypting comes from the attacker, and is thus safe to output.

To do this, we annotate the key k generated in process S with the following type:

$$T_k = \text{seskey}^{\mathtt{HH},1}((\mathtt{HL} * \mathtt{HL} * \tau_{r'_1}^{\mathtt{HH},1}) \vee (\mathtt{HL} * \mathtt{HL} * \tau_{r'_2}^{\mathtt{HH},1}))$$

This type specifies that k is a trusted session key (label HH), that is used by honest agents (*i.e.* Alice and Bob) to encrypt tuples whose last component is either r_1 or r_2. Of course, dishonest agent (*i.e.* the attacker) may also use it to encrypt any value (which will thus be of type LL). This type will be helpful when typechecking the weeding phase, as it gives information regarding the value of the variable being compared, *i.e.* the last component of the tuple.

We also annotate the nonces r'_1, r'_2 respectively with types $\tau_{r'_1}^{\mathtt{HH},1}$, $\tau_{r'_2}^{\mathtt{HH},1}$ indicating they are secret. The nonces r_1, r_2 are respectively annotated with $\tau_{r_1}^{\mathtt{LL},1}$ and $\tau_{r_2}^{\mathtt{LL},1}$, indicating they may be published (as they are sent in clear on the network).

We then type the protocol using the initial environment Γ below.

[6] 0 and 1 are represented as public constants in our model

$$\begin{aligned}
\Gamma(k_1, k_1)) &= \mathrm{eqkey}^{\mathtt{HH}}(\{[\![\tau_0^{\mathtt{LL},1}\,;\,\tau_1^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r_1'}^{\mathtt{HH},1}\}_{\mathrm{pkey}(T_k)}) \\
\Gamma(k_2, k_2) &= \mathrm{eqkey}^{\mathtt{HH}}(\{[\![\tau_1^{\mathtt{LL},1}\,;\,\tau_0^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r_2'}^{\mathtt{HH},1}\}_{\mathrm{pkey}(T_k)}) \\
\Gamma(k_0, k_0) &= \mathrm{eqkey}^{\mathtt{HH}}(\mathrm{pkey}(T_k)) \\
\Gamma(k', k') &= \mathrm{eqkey}^{\mathtt{HH}}(\tau_{r_1'}^{\mathtt{HH},1} * \tau_{r_2'}^{\mathtt{HH},1} * T_k)
\end{aligned}$$

Since, in this protocol, we do not use different keys on either side, we give all keys an $\mathrm{eqkey}^{\mathtt{HH}}(\cdot)$ type. The type of k_1 states that it signs encryptions of vote 0 on the left and 1 on the right, together with the nonce r_1', with a key of type $\mathrm{pkey}(T_k)$. Similarly, the type of k_2 states it signs encryptions of 1 on the left and 0 on the right. Indeed, these keys are only used by honest voters to encrypt their votes with k (that they received from the network). The type of k_0 states that it signs a key of type $\mathrm{pkey}(T_k)$: indeed in practice it signs $\mathtt{pk}(k)$. This type will let us know when typechecking V that the key received signed with k_0 has indeed the correct type. Finally the type of the key k' used for internal communication between the two parts of the server states that it encrypts r_1', r_2', and k (or rather, a key with the same type).

We focus first on the voter process for Alice, *i.e.* $V(k_1, 0) \sim V(k_1, 1)$. We start by applying rule PIn, which requires us to typecheck the rest of the process with $x : \mathtt{LL}$. Rule PLet they lets us typecheck the signature verification $\mathtt{checksign}(x, \mathtt{vk}(k_0))$: we need to typecheck the two then-branches, and the two else-branches. The else-branches are 0 and trivially typecheck, without generating constraints[7]. In the then-branches, the type of k_0 in Γ indicates that x_k has type $\mathrm{pkey}(T_k)$. Rule PIn then applies, followed by rule PNew, and these require us to typecheck the continuation process with $x_r : L$ and $r_1' : \tau_{r_1'}^{\mathtt{HH},1}$. We then apply rule POut to typecheck the last instruction, which is the output of the ballot. We must give type $\mathtt{LL}$ to the messages, *i.e.* prove

$$\Gamma' \vdash \mathtt{sign}(\mathtt{aenc}(\langle 0, \langle x_r, r' \rangle\rangle, x_k), k) \sim \mathtt{sign}(\mathtt{aenc}(\langle 1, \langle x_r, r' \rangle\rangle, x_k), k) : \mathtt{LL}$$

where Γ' is Γ extended with types for $x, x_k, etc.$ as stated above. Figure 3.1 displays the type derivation for this. Typing the ballot produces constraint

$$c = \{M \sim N, \mathtt{sign}(M, k_1) \sim \mathtt{sign}(N, k_1)\}$$

where $M = \mathtt{aenc}(\langle 0, \langle x_r, r_1' \rangle\rangle, x_k)$ and $N = \mathtt{aenc}(\langle 1, \langle x_r, r_1' \rangle\rangle, x_k)$.

Rule PZero ends the typing of the voting process for Alice, which thus produces constraint

$$C_a = \{(c, \Gamma')\}$$

for the c and Γ' defined above.

The voter process for Bob is typed similarly as the one for Alice[8], and produces

$$C_b = \{(c', \Gamma'')\}$$

where c' is similar to c, except that the votes are swapped and are signed with k_2 instead of k_1, and Γ'' is similar to Γ', with the nonce r_2' instead of r_1'.

[7] They actually generate constraint sets containing only the empty set of constraints. We omit these from the presentation, for clarity, as they do not matter for consistency.

[8] In Bob's process, variables have to be renamed to avoid collisions with the variables in Alice's. We omit this detail here.

$$*_1 = \dfrac{\dfrac{}{\Gamma' \vdash 0 \sim 1 : [\![\tau_0^{\mathtt{LL},1} ; \tau_1^{\mathtt{LL},1}]\!] \to \emptyset}\,\text{TLR}^1 \qquad \dfrac{}{\Gamma' \vdash x_r \sim x_r : \mathtt{HL} \to \emptyset}\,\text{THigh} \qquad \dfrac{\Gamma'(r_1') = \tau_{r_1'}^{\mathtt{HH},1}}{\Gamma' \vdash r_1' \sim r_1' : \tau_{r_1'}^{\mathtt{HH},1} \to \emptyset}\,\text{TLR}^1}{\Gamma' \vdash \langle 0, \langle x_r, r_1' \rangle\rangle \sim \langle 1, \langle x_r, r_1' \rangle\rangle : T \to \emptyset}\,\text{TPair x2}$$

$$*_2 = \dfrac{\dfrac{*_1}{\Gamma' \vdash \langle 0, \langle x_r, r_1' \rangle\rangle \sim \langle 1, \langle x_r, r_1' \rangle\rangle : T \to \emptyset} \qquad \dfrac{\Gamma'(x_k) = \text{pkey}(T_k)}{\Gamma' \vdash x_k \sim x_k : \text{pkey}(T_k) \to \emptyset}\,\text{TVar}}{\Gamma' \vdash M \sim N : \{T\}_{\text{pkey}(T_k)} \to \emptyset}\,\text{TAenc}$$

$$*_3 = \dfrac{\dfrac{\dfrac{}{\Gamma' \vdash 0 \sim 1 : \mathtt{HL} \to \emptyset}\,\text{THigh} \qquad \dfrac{}{\Gamma' \vdash x_r \sim x_r : \mathtt{HL} \to \emptyset}\,\text{THigh} \qquad \dfrac{\Gamma'(r_1') = \tau_{r_1'}^{\mathtt{HH},1}}{\Gamma' \vdash r_1' \sim r_1' : \tau_{r_1'}^{\mathtt{HH},1} \to \emptyset}\,\text{TLR}^1}{\Gamma' \vdash \langle 0, \langle x_r, r_1' \rangle\rangle \sim \langle 1, \langle x_r, r_1' \rangle\rangle : \mathtt{HL} * \mathtt{HL} * \tau_{r_1'}^{\mathtt{HH},1} \to \emptyset}\,\text{TPair x2}}{\Gamma' \vdash \langle 0, \langle x_r, r_1' \rangle\rangle \sim \langle 1, \langle x_r, r_1' \rangle\rangle : T' \to \emptyset}\,\text{TOr}$$

$$*_4 = \dfrac{\dfrac{\dfrac{*_3}{\Gamma' \vdash \langle 0, \langle x_r, r_1' \rangle\rangle \sim \langle 1, \langle x_r, r_1' \rangle\rangle : T' \to \emptyset} \qquad \dfrac{\Gamma'(x_k) = \text{pkey}(T_k)}{\Gamma' \vdash x_k \sim x_k : \text{pkey}(T_k) \to \emptyset}\,\text{TVar}}{\Gamma' \vdash M \sim N : \{T'\}_{\text{pkey}(T_k)} \to \emptyset}\,\text{TAenc}}{\Gamma' \vdash M \sim N : \mathtt{LL} \to \{M \sim N\}}\,\text{TAencH}$$

$$\dfrac{\dfrac{*_2}{\Gamma' \vdash M \sim N : \{T\}_{\text{pkey}(T_k)} \to \emptyset} \qquad \dfrac{*_4}{\Gamma' \vdash M \sim N : \mathtt{LL} \to c_2} \qquad \Gamma(k_1, k_1) = \text{eqkey}^{\mathtt{HH}}(T)\text{pkey}(T_k)}{\Gamma' \vdash \mathtt{sign}(M, k_1) \sim \mathtt{sign}(N, k_1) : \mathtt{LL} \to c}\,\text{TSignH}$$

where
$c = \{M \sim N, \mathtt{sign}(M, k_1) \sim \mathtt{sign}(N, k_1)\}$,
$M = \mathtt{aenc}(\langle 0, \langle x_r, r_1' \rangle\rangle, x_k)$, $N = \mathtt{aenc}(\langle 1, \langle x_r, r_1' \rangle\rangle, x_k)$,
$T = [\![\tau_0^{\mathtt{LL},1} ; \tau_1^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r_1'}^{\mathtt{HH},1}$,
$T' = (\mathtt{HL} * \mathtt{HL} * \tau_{r_1'}^{\mathtt{HH},1}) \vee (\mathtt{HL} * \mathtt{HL} * \tau_{r_2'}^{\mathtt{HH},1})$.

Figure 3.1 – Type derivation for the ballot of Alice

We then proceed with the server process S. The beginning is rather straightforward: rule PNewKey stores $k : T_k$ in the environment, rule POut then requires that we give type LL to the message containing the election public key has type LL. This can be done without generating any constraints using rule TSignL. The same goes for the output of the two random values r_1, r_2. Then rule PIn assigns type LL to x_1, x_2, and we proceed to the signature verification.

The verification is typechecked using rule PLet, and the type of k_1 inform us that y_1 has type $\{[\![\tau_0^{\mathtt{LL},1} ; \tau_1^{\mathtt{LL},1}]\!] * \mathtt{HL} * \tau_{r_1'}^{\mathtt{HH},1}\}_{\text{pkey}(T_k)}$. Using rule PLet again (with destructor rule DAdecT'), we get in particular that z_{v1} has type $[\![\tau_0^{\mathtt{LL},1} ; \tau_1^{\mathtt{LL},1}]\!]$. Similarly, after checking the signature for Bob's ballot and decrypting it, we get that z_{v2} has type $[\![\tau_1^{\mathtt{LL},1} ; \tau_0^{\mathtt{LL},1}]\!]$.

We then only have to typecheck the publication of the votes in clear. We then use rule PPar to typecheck the two parallel outputs separately. As explained earlier, we swap these two outputs on the right, so that we have to typecheck $\mathtt{out}(z_{v1}) \sim \mathtt{out}(z_{v2})$ and $\mathtt{out}(z_{v2}) \sim \mathtt{out}(z_{v1})$. These two branches are similar, we describe the first one. By rule POut, it requires us to give type LL to $z_{v1} \sim z_{v2}$. This is done using rule TLRVar, which allows us to use the two variables' refinement types to give type $[\![\tau_0^{\mathtt{LL},1} ; \tau_0^{\mathtt{LL},1}]\!]$ to them. Then rule TLRL' lets us give them type LL, without generating any constraints.

The server process for honest voters S thus typechecks without generating any constraints.

We now typecheck the second part of the server process S'. Again, the beginning is easy: rule

PIN assigns type LL to y, and following the type of k', rule PLETDEC[9] for decryption assigns types $\tau^{\mathtt{HH},1}_{r'_1}$, $\tau^{\mathtt{HH},1}_{r'_2}$ and T_k respectively to y_{r1}, y_{r2}, y_k. The dishonest ballot y_3 is given type LL again by rule PIN. The intricate point is the decryption of this ballot and weeding. At this point, the ballot received could be either a copy of Alice's, a copy of Bob's, or a ballot generated by the attacker. Rule PLET (with destructor rule DADECH') reflects these three cases: since we decrypt with a variable with type T_k, we need to typecheck the continuation process three times, which will correspond to each of these three cases, and then combine all constraints obtained. More precisely, there are three possible types for variables z_{v3}, z_{r3}, z'_{r3}: two from the two branches of the union type in the payload type of T_k (which we split using POR), and one for the case where the ciphertext is produced by the attacker. Formally we have to typecheck the rest of the process with the following types for variables:

- once with $z_{v3} : \mathtt{HL}, z_{r3} : \mathtt{HL}, z'_{r3} : \tau^{\mathtt{HH},1}_{r'_1}$. Then when performing the first equality check, rule PIFLR can use the refinement types of z'_{r3} and y_{r1} to show that the test always succeeds. We thus only have to typecheck the then-branch, which is 0 and trivially typechecks with no constraints.
- once with $z_{v3} : \mathtt{HL}, z_{r3} : \mathtt{HL}, z'_{r3} : \tau^{\mathtt{HH},1}_{r'_2}$. This time, rule PIFLR similarly shows that the first test fails, and the second (*i.e.* $z'_{r3} = y_{r2}$) succeeds. Similarly, we thus only have to typecheck the second then-branch, which is 0.
- once with $z_{v3} : \mathtt{LL}, z_{r3} : \mathtt{LL}, z'_{r3} : \mathtt{LL}$. The refinements of both y_{r1} and y_{r2} allow us to give them type HH (without constraints) using rule TLR'. Using this information, rule PIFS detects that, since z'_{r3} has type LL, both equality checks fail, and we only have to typecheck the last else-branch, *i.e.* the output of z_{v3}. This is easily done using POUT, without creating any constraints, since this variable has type LL.

Notice that the typechecking of the weeding entirely relies on the use of refinements. Indeed, in the first two cases, z_{v3} has type HL, and thus cannot be output. It has to be this way, since, intuitively, in that case, this variable contains the vote of Alice or Bob, which is different on the left and on the right. It is therefore crucial for the typechecking that in the first two case we can statically detect that the weeding removes this ballot, and that there is no need to typecheck the output of z_{v3}. This is only possible thanks to the refinements of variables z'_{r3}, y_{r1}, and y_{r2}.

In addition, the *Setup* process just consists in the output of public values, which can easily be given type LL without any constraints. It then clearly typechecks with an empty constraint set using rule POUT.

Altogether, after applying rule PPAR, the constraint set for the non-replicated part of the process, *i.e.* for $\mathit{Setup} \mid V(k_1, v_1) \mid V(k_2, v_2) \mid S(\mathtt{vk}(k_1), \mathtt{vk}(k_2))$, is $C = C_a \cup_\times C_b$. That is to say, omitting from the environment all variables and names that do not occur in the messages:

$$C = \{ \quad (\{M \sim N, \mathtt{sign}(M, k_1) \sim \mathtt{sign}(N, k_1), M' \sim N', \mathtt{sign}(M', k_2) \sim \mathtt{sign}(N', k_2)\}, \\ [x_r : \mathtt{LL}, x_k : T_k, x'_r : \mathtt{LL}, x'_k : T_k, r'_1 : \tau^{\mathtt{HH},1}_{r'_1}, r'_2 : \tau^{\mathtt{HH},1}_{r'_2}]) \quad \}$$

where $M = \mathtt{aenc}(\langle 0, \langle x_r, r'_1\rangle\rangle, x_k)$ and $N = \mathtt{aenc}(\langle 1, \langle x_r, r'_1\rangle\rangle, x_k)$ are Alice's ballots, and $M' = \mathtt{aenc}(\langle 1, \langle x'_r, r'_2\rangle\rangle, x'_k)$ and $N' = \mathtt{aenc}(\langle 0, \langle x'_r, r'_2\rangle\rangle, x'_k)$ are Bob's[10].

[9] Since the key k' is used only for this purpose, it is never matched with other keys in Γ, and thus in that case rule PLETDEC does not create many proof obligations.

[10] As stated earlier, we renamed the variables x_r and x_k in Bob's ballots.

The replicated part of the process, *i.e.* S', does not generate any constraints when typechecked. Therefore, we only need to check the consistency of C.

3.2.3 Consistency for Helios, and conclusion

We do not detail entirely the application of the procedure `check_const` to C here, but only the most relevant points.

- `step1` actually leaves the constraints unchanged, since no variables in the messages have refinement types.
- `step2` is satisfied on all elements in C: in particular, all encryptions are with honest keys and contain the nonce r'_1 or r'_2, whose types have label `HH`.
- `step3` is satisfied on all elements in C, as none of the messages M, M', $\mathtt{sign}(M, k_1)$, $\mathtt{sign}(M', k_2)$ (resp. N, N', $\mathtt{sign}(N, k_1)$, $\mathtt{sign}(N', k_2)$ on the right) are unifiable with one another. Indeed, M and M' contain different votes (0 and 1 respectively), and thus cannot be unified, and similarly for the two signatures; and the encryptions M and M' of course cannot be unified with the signatures. Hence, the `check_const` procedure applied to C succeeds. It also clearly succeeds on $[\, C \,]_1$ which is just a renaming of C.

Therefore, Theorem 7 proves that Helios indeed satisfies the vote privacy property, when considering two honest voters Alice and Bob, and an unbounded number of dishonest voters.

Note that this proof only holds thanks to the fact that Alice and Bob each vote only once. Having Alice vote several times would require the election public key to have a more general type, that allows it to encrypt several different ballots from Alice. When typechecking the decryption performed by the server S', we could then not discard the case where the ballot received is a copy of Alice's: we cannot statically know that the weeding test will succeed (*i.e.*, that the ballot will be rejected) in that case. In fact, as already mentioned, there is an attack (discovered by Roenne) in case voters revote. The attacker may block one of Alice's ballots, and submit a copy of it in his name, which could not be detected by weeding, as the server has never seen it. This attack corresponds exactly to the case we could not typecheck, as we just described. However this attack is not possible if voters do not revote: our type system is able to leverage this by encoding in the types the information that each voter can only produce one ballot.

3.3 Private Authentication

We now show how our type system can be applied to type the Private Authentication protocol evoked in Section 2.2, by showing the most interesting parts of the derivation.

3.3.1 Modelling anonymity for Private Authentication

Recall that the protocol was informally described as follows

$$
\begin{aligned}
A \to B :&\quad \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, \mathtt{pk}(k_b)) \\
B \to A :&\quad \begin{cases} \mathtt{aenc}(\langle N_a, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) & \text{if } B \text{ accepts } A\text{'s request} \\ \mathtt{aenc}(N_b, \mathtt{pk}(k)) & \text{if } B \text{ declines } A\text{'s request} \end{cases}
\end{aligned}
$$

where $\mathtt{aenc}(N_b, \mathtt{pk}(k))$ is a decoy message indicating that B refuses to communicate with A, and $\mathtt{pk}(k)$ is a decoy key (no one knows the private key k).

As briefly mentioned in Section 2.2, we model the role of the initiator A with key k_a trying to authenticate an agent with public key pk_b as the following process:

$$P_a(k_a, pk_b) = \mathtt{new}\ N_a.\mathtt{out}(\mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b)).\ \mathtt{in}(z)$$

We model the role of agent B with key k_a willing to authenticate to an agent with public key pk_b (and refusing requests from other agents) as the following process $P_b(k_b, pk_b)$:

$$\begin{aligned} P_b(k_b, pk_a) \quad = \quad & \mathtt{new}\ N_b.\ \mathtt{in}(x). \\ & \mathtt{let}\ y = \mathtt{adec}(x, k_b)\ \mathtt{in}\ \ \mathtt{let}\ y_1 = \pi_1(y)\ \mathtt{in}\ \ \mathtt{let}\ y_2 = \pi_2(y)\ \mathtt{in} \\ & \quad \mathtt{if}\ y_2 = pk_a\ \mathtt{then} \\ & \quad\quad \mathtt{out}(\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, pk_a)) \\ & \quad \mathtt{else}\ \mathtt{out}(\mathtt{aenc}(N_b, \mathtt{pk}(k))) \end{aligned}$$

The protocol aims to provide anonymity, that is, intuitively, an attacker should not be able to tell whether B is willing to accept authentication requests from agent A, or rejects them and is only willing to communicate with another honest[11] agent C[12]. Following the model proposed in [14], we formalise this property as the following equivalence:

$$P_a(k_a, \mathtt{pk}(k_b))\ \mid\ P_b(k_b, \mathtt{pk}(k_a)) \ \approx_t\ P_a(k_a, \mathtt{pk}(k_b))\ \mid\ P_b(k_b, \mathtt{pk}(k_c))$$

3.3.2 Typechecking Private Authentication

The key point when proving this equivalence is that the execution is not uniform. The authentication request can typically succeed on the left and fail on the right in the corresponding execution. In that case, the real response, encrypted with k_b, is output in the left process, while the decoy response, encrypted with the decoy key k, is output on the right. For each possible such case, where a different key is used on either side, the typing environment must contain a binding for the corresponding bikey.

We type the protocol using the initial environment Γ presented earlier in Figure 2.1, and reproduced here.

$\Gamma(k_b, k_b)$	$=$	$\mathrm{key}^{\mathtt{HH}}(\mathtt{HL} * \mathtt{LL})$	initial message uses same key on both sides
$\Gamma(k_a, k)$	$=$	$\mathrm{key}^{\mathtt{HH}}(\mathtt{HL})$	authentication succeeded on the left, failed on the right
$\Gamma(k, k_c)$	$=$	$\mathrm{key}^{\mathtt{HH}}(\mathtt{HL})$	authentication succeeded on the right, failed on the left
$\Gamma(k_a, k_c)$	$=$	$\mathrm{key}^{\mathtt{HH}}(\mathtt{HL})$	authentication succeeded on both sides
$\Gamma(k, k)$	$=$	$\mathrm{key}^{\mathtt{HH}}(\mathtt{HL})$	authentication failed on both sides

We also annotate the declarations of N_a and N_b respectively with type $\tau_{N_a}^{\mathtt{HH},1}$ and $\tau_{N_b}^{\mathtt{HH},1}$, specifying they should be secret.

We of course first apply rule PPar, to typecheck independently the initiator and responder processes, *i.e.* $P_a(k_a, \mathtt{pk}(k_b)) \sim P_a(k_a, \mathtt{pk}(k_b))$ and $P_b(k_b, \mathtt{pk}(k_a) \sim P_b(k_b, \mathtt{pk}(k_c))$.

[11] If C were dishonest, the attacker would know his private key and could simply try to send a request with it to see who B is willing to talk to.

[12] Note that this is possible even though the decoy message and the true response have different length since, as mentioned in Chapter 2, we model length-hiding encryption.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{* = \langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, N_b \text{ well formed}}
}{\Gamma'' \vdash \langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle \sim N_b : \mathtt{HL} \to \emptyset}\ \textsc{THigh}
\qquad
\dfrac{\Gamma(k_a, k) = \mathrm{key}^{\mathtt{HH}}(\mathtt{HL})}{\Gamma'' \vdash k_a \sim k : \mathrm{key}^{\mathtt{HH}}(\mathtt{HL}) \to \emptyset}\ \textsc{TKey}
}{\Gamma'' \vdash \mathtt{pk}(k_a) \sim \mathtt{pk}(k) : \mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL})) \to \emptyset}\ \textsc{TPubKey}
}{\Gamma'' \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)) : \{\mathtt{HL}\}_{\mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL}))} \to \emptyset}\ \textsc{TAenc}
}{\Gamma'' \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k)) : \mathtt{LL} \to c}\ \textsc{TAencH}
$$

where $c = \{\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))\}$.

Figure 3.2 – Type derivation for the response to A and the decoy message

We focus first on the responder process P_b. We start by applying rule PNew, which simply adds to Γ the binding $N_b : \tau_{N_b}^{\mathtt{HH},1}$. Then, rule PIn requires us to typecheck the rest of the process with $x : \mathtt{LL}$. Let us write $\Gamma' = \Gamma, x : \mathtt{LL}, N_b : \tau_{N_b}^{\mathtt{HH},1}$.

We then proceed with the asymmetric decryption. As we use the same key k_b in both processes, we apply rule PLetAdecSame. We have $\Gamma'(x) = \mathtt{LL}$ from rule PIn and $\Gamma'(k_b, k_b) = \mathrm{key}^{\mathtt{HH}}(\mathtt{HL}{*}\mathtt{LL})$. We do not have any other entry using key k_b in Γ'. We hence need to typecheck the two `then` branches once with $\Gamma', y : (\mathtt{HL} * \mathtt{LL})$ and once with $\Gamma', y : \mathtt{LL}$, as well as the two `else` branches (which are just 0 in this case, and are trivially typed using rule PZero).

Typing the let expressions where the message in y is projected onto its two components is straightforward using rule PLet. When typechecking with $y : \mathtt{LL}$, they specify that the continuation must be typechecked with $\Gamma', y : \mathtt{LL}, y_1 : \mathtt{LL}, y_2 : \mathtt{LL}$. Otherwise, when $y : (\mathtt{HL} * \mathtt{LL})$, it must be typechecked with $\Gamma', y : (\mathtt{HL} * \mathtt{LL}), y_1 : \mathtt{HL}, y_2 : \mathtt{LL}$.

In the conditional we check $y_2 = \mathtt{pk}(k_a)$ in the left process and $y_2 = \mathtt{pk}(k_c)$ in the right process. In any case, we only know at this point that y_2 has type $\mathtt{LL}$. This variable is compared to messages $\mathtt{pk}(k_a)$ and $\mathtt{pk}(k_c)$, which on the contrary cannot be given type $\mathtt{LL}$, as observing them would let the attacker distinguish between the two sides. Hence we cannot guarantee which branches are taken, or even if the same branch is taken in the two processes. We therefore use rule PIfAll to typecheck all four possible combinations of branches. Note that in doing this we also typecheck cases where the authentication succeeds on the right, which should not happen in real executions, as the agent C that B accepts requests from is not present. This is expected: when typechecking, we overapproximate the behaviours of the processes. With finer typing rules, we might be able to statically determine that such cases cannot happen, producing less constraints to check for consistency. Since the constraints produced here are consistent anyway, this is not an issue here, and we can still prove the property despite this overapproximation.

We now focus on the case where A's request is successful in the left process and is rejected in the right process. We then have to typecheck B's positive answer together with the decoy message: $\Gamma'' \vdash \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_c, \mathtt{pk}(k)) : \mathtt{LL}$, where Γ'' is either $\Gamma''_1 = \Gamma', y_1 : \mathtt{HL}, y_2 : \mathtt{LL}$, or $\Gamma''_2 = \Gamma', y_1 : \mathtt{LL}, y_2 : \mathtt{LL}$, depending on which of the two proof obligations created by rule PLetAdecSame we are proving.

Figure 3.2 presents the type derivation for this example. We apply rule TAenc to give type $\mathtt{LL}$ to the two terms, adding the two encryptions to the constraint set. Using rule TAencH we can show that the encryptions are well-typed with type $\{\mathtt{HL}\}_{\mathrm{pkey}(\mathrm{key}^{\mathtt{HH}}(\mathtt{HL}))}$. The type of the payload is trivially shown with rule THigh. To type the public key, we use rule TPubKey followed by rule TKey, which looks up the type for the bikey (k_a, k) in the typing environment Γ''.

The other three cases can be typechecked similarly, regardless of the type of y_1, yielding

each time a constraint containing a single pair of messages: either two honest responses, a decoy message and a honest response, or two decoy messages.

Altogether, typechecking the responder process thus yields the constraint set $C_b = C_{b,1} \cup C_{b,2}$, where[13]

$$\begin{aligned} C_{b,1} = \{ \quad & (\{\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_c))\}, \Gamma_1''), \\ & (\{\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))\}, \Gamma_1''), \\ & (\{\mathtt{aenc}(N_b, \mathtt{pk}(k)) \sim \{\mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_c))\}, \Gamma_1''), \\ & (\{\mathtt{aenc}(N_b, \mathtt{pk}(k)) \sim \mathtt{aenc}(N_b, \mathtt{pk}(k))\}, \Gamma_1'') \quad \} \end{aligned}$$

and $C_{b,2}$ is identical to $C_{b,1}$ except its elements contain Γ_2'' instead of Γ_1''.

The initiator process P_a is more easily typechecked: as for the responder, rule PNew first simply adds $N_a : \tau_{N_a}^{\mathtt{HH},1}$ to the environment. Then we use rule POut, which requires to give type $\mathtt{LL}$ to the request message, *i.e.* $\Gamma, N_a : \tau_{N_a}^{\mathtt{HH},1} \vdash \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b) \sim \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b) : \mathtt{LL}$.

This is done in the same way as the response message, and yields constraint

$$\mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b) \sim \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b).$$

The rest of the process trivially typechecks with rules PIn and PZero, and the initiator process thus typechecks with constraint

$$C_a = \{(\{\mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b) \sim \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b)\}, (\Gamma, N_a : \tau_{N_a}^{\mathtt{HH},1}, z : \mathtt{LL}))\}.$$

Altogether, after applying rule PPar, the constraint set for the whole process is thus $C = C_a \cup_{\times} C_b$, *i.e.*

$$\begin{aligned} C = \{ \quad & (\{Q \sim Q, R \sim R'\}, \Gamma_1), (\{Q \sim Q, R \sim D\}, \Gamma_1), \\ & (\{Q \sim Q, D \sim R'\}, \Gamma_1), (\{Q \sim Q, D \sim D\}, \Gamma_1), \\ & (\{Q \sim Q, R \sim R'\}, \Gamma_2), (\{Q \sim Q, R \sim D\}, \Gamma_2), \\ & (\{Q \sim Q, D \sim R'\}, \Gamma_2), (\{Q \sim Q, D \sim D\}, \Gamma_2) \quad \} \end{aligned}$$

where $Q = \mathtt{aenc}(\langle N_a, \mathtt{pk}(k_a)\rangle, pk_b)$ is the request, $R = \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_a))$ and $R' = \mathtt{aenc}(\langle y_1, \langle N_b, \mathtt{pk}(k_b)\rangle\rangle, \mathtt{pk}(k_c))$ are the responses to A and C respectively, and lastly $D = \mathtt{aenc}(N_b, \mathtt{pk}(k))$ is the decoy message. Γ_1 is the environment

$$\Gamma, N_a : \tau_{N_a}^{\mathtt{HH},1}, N_b : \tau_{N_b}^{\mathtt{HH},1}, x : \mathtt{LL}, y : \mathtt{HL} * \mathtt{LL}, y_1 : \mathtt{HL}, y_2 : \mathtt{LL}, z : \mathtt{LL}$$

and Γ_2 is identical except that $\Gamma_2(y) = \Gamma_2(y_1) = \mathtt{LL}$.

3.3.3 Consistency for Private Authentication, and conclusion

We do not detail entirely the application of the procedure `check_const` to C here, but only the most relevant points.

- `step1` is actually the identity function here, as neither Γ_1 nor Γ_2 contain refinement types: it leaves the constraints unchanged.

[13] The actual constraint set also contains empty sets of constraints generated when typechecking the else-branch corresponding to the case where decryption or projection fails on both sides. We omit them here for clarity, as they do not matter for consistency.

- `step2` is satisfied on all elements in C: in particular, all encryptions are with honest keys and contain the nonce N_b whose type is $\tau_{N_b}^{\mathtt{HH},1}$ in both Γ_1 and Γ_2.
- `step3` is satisfied on all elements in C, as none of the messages Q, R, D are unifiable with one another (and similarly on the right for Q, R', D).

Therefore, the soundness theorems (Theorems 1 and 3) prove that Private Authentication indeed satisfies the anonymity property, when considering one session of the initiator and one session of the responder.

We may even use Theorem 7 to consider an unbounded number of sessions of both the initiator and the responder, and prove that

$$!\,(P_a(k_a, \mathtt{pk}(k_b)) \;\mid\; P_b(k_b, \mathtt{pk}(k_a))) \;\approx_t\; !\,(P_a(k_a, \mathtt{pk}(k_b)) \;\mid\; P_b(k_b, \mathtt{pk}(k_c))).$$

According to the theorem, we still need to typecheck only one session to prove this equivalence: we have already done this above. The names N_a and N_b now have infinite types $\tau_{N_a}^{\mathtt{HH},\infty}$ and $\tau_{N_b}^{\mathtt{HH},\infty}$, but this has no influence on the typechecking since no rules where this distinction matters were used.

As there is no unreplicated part here, what remains to be done is simply to ensure that

$$\mathtt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2) = \mathtt{true}.$$

The reasoning for this is the same as in the bounded case for the first two steps. For the third step, the only new unifications that could potentially happen would be between the two (renamed) copies of Q, the two copies of R (or R'), or the two copies of D. However, all of these messages contain one of the nonces N_a, N_b. Since these now have infinite types $\tau_{N_a}^{\mathtt{HH},\infty}$ and $\tau_{N_b}^{\mathtt{HH},\infty}$, they are renamed in each of the copies, and therefore prevent these unifications. Therefore the procedure succeeds, and Theorem 7 proves the equivalence above: Private authentication satisfies anonymity with an unbounded number of sessions of both the initiator and the responder.

3.4 Experimental Results

We have implemented in collaboration with Niklas Grimm a prototype type-checker for our type system, which we will call TypeEq in the following, and applied it on various examples briefly described below. Its source code, as well as the files from the examples, are available at [1].

Symmetric key protocols For the sake of comparison, we consider two symmetric key protocols taken from the benchmark of [55], and described in [53]: Needham-Schroeder and Yahalom-Lowe. These protocols aim at exchanging a key symmetric k. In both cases, we prove key usability of the exchanged key. Intuitively, we show that an attacker cannot distinguish between two encryptions of public constants: $P.\mathtt{out}(\mathtt{enc}(a, k)) \;\approx_t\; P.\mathtt{out}(\mathtt{enc}(b, k))$. Note in particular that this equivalence is immediately broken if the attacker is able to learn k.

Asymmetric key protocols In addition to the symmetric key protocols, we consider the well-known Needham-Schroeder-Lowe (NSL) protocol [89] and we again prove strong secrecy of the nonce sent by the receiver (Bob).

Helios We model the Helios protocol for two honest voters and infinitely many dishonest ones, as described in Section 3.2. As emphasised in Section 2.2, Helios is secure only if honest voters vote at most once. Therefore the protocol includes non replicated processes (for voters) as well as a replicated process (to handle dishonest ballots).

BAC The Basic Access Control (BAC) protocol is one of the protocols embedded in the biometric passport [5]. We show anonymity of the passport holder $P(A) \approx_t P(B)$. Actually, the only data that distinguishes $P(A)$ from $P(B)$ are the private keys. Therefore, to make the property non-trivial, we consider an additional step where the passport sends the identity of the agent to the reader, encrypted with the exchanged key.

Private Authentication Finally, we consider the private authentication protocol, as described in the previous section.

All our experiments have been run on a single Intel Xeon E5-2687Wv3 3.10GHz core, with 378GB of RAM (shared with the 19 other cores). Actually, our own prototype does not require a large amount of RAM. However, some of the other tools we consider used more than 64GB of RAM on some examples (at which point we stopped the experiment).

3.4.1 Bounded number of sessions

We first compare our tool with tools designed for a bounded number of sessions. We ran our benchmarks using the latest public versions at the time of writing of SPEC [68], APTE (and its APTE-POR variant) [45, 18], Akiss [43], SAT-Equiv [55], and Deepsec [48]. The protocol models may slightly differ due to the subtleties of each tool. For example, several of these tools require *simple* processes where each sub-process emits on a distinct channel. We do not need such an assumption. Moreover, Helios involves non-trivial else branches, which are not supported by all of the tools.

The number of sessions we consider denotes the number of processes (modelling protocol roles) in parallel in each scenario. For symmetric key protocols, we start with a simple scenario with only two honest participants A, B and a honest server S (3 sessions). We consider increasingly more complex scenarios (6, 10, and 14 sessions) featuring a dishonest agent C.

In the complete scenario (14 sessions) each agent among A, B (and C) runs the protocol once as the initiator, and once as the responder with each other agent (A, B, C). In the case of NSL, we similarly consider a scenario with two honest agents A, B running the protocol once (2 sessions), and two scenarios with an additional dishonest agent C, up to the complete scenario (8 sessions) where each agent runs NSL once as initiator, once as responder, with each agent. In some cases, we further increase the number of sessions (replicating identical scenarios) to better compare tools performance. For Helios, we consider 2 honest voters, and one dishonest voter only, as well as a ballot box. The corresponding results are reported in Figure 3.3.

In several cases the tools were not able to prove the protocol, for different reasons. We write TO for Time Out (we stopped the experiments after 12 hours), MO for Memory Out (we stopped the experiments when the tool used up more than 64 GB of RAM), SO for Stack Overflow, BUG in the case of APTE, when the proof failed due to bugs in the tool, and x when the tool could not handle the protocol for the reasons discussed previously. In all cases, our tool is almost instantaneous and outperforms by orders of magnitude the competitors.

Protocols (# sessions)		Akiss	APTE	APTE-POR	SPEC	SAT-Equiv	Deepsec	TypeEq	
								Time	Memory
Needham - Schroeder (symmetric)	3	4.2s	0.39s	0.086s	59.3s	0.05s	0.007s	0.006s	4.0 MB
	6	TO	TO	9m22s	TO	0.27s	0.017s	0.009s	4.7 MB
	10			SO		1.9s	0.10s	0.012s	5.0 MB
	14					10s	19s	0.015s	6.9 MB
Yahalom - Lowe	3	1.0s	2.9s	0.095s	10s	0.040s	0.008s	0.006s	3.8 MB
	6	MO	TO	11m20s	MO	0.20s	0.021s	0.017s	4.9 MB
	10			SO		3.2s	0.92s	0.015s	4.9 MB
	14					20s	3m12s	0.019s	5.0 MB
Needham-Schroeder-Lowe	2	0.10s	3.8s	0.06s	28s	0.051s	0.007s	0.004s	3.1 MB
	4	1m8s	BUG	BUG	TO	0.13s	0.010s	0.005s	3.4 MB
	8	TO				54s	0.547s	0.009s	4.7 MB
Private Authentication	2	0.19s	1.2s	0.034s	x	x	0.012s	0.004s	3.2 MB
	4	99m	TO	24.6s			18s	0.013s	4.9 MB
	8	MO		TO			TO	1s	37 MB
Helios	3	MO	BUG	BUG	x	x	TO	0.005s	3.5 MB
BAC	2	4.0s	0.20s	0.032s	x	x	0.016s	0.004s	2.9 MB
	3	SO	185m	2.6s			50s	0.004s	3.1 MB
	5		TO	107m			TO	0.005s	3.4 MB
	7			TO				0.006s	3.8 MB

TO: Time Out (>12h) MO: Out of Memory (>64GB) SO: Stack Overflow

Figure 3.3 – Experimental results for the bounded case

3.4.2 Unbounded numbers of sessions

We then compare our type-checker with the latest public version of ProVerif [35] at the time of writing, for an unbounded number of sessions, on a the same examples as for the bounded case. As expected, ProVerif cannot prove Helios secure since it cannot express that voters vote only once. This may sound surprising, since proofs of Helios in ProVerif already exist (*e.g.* [65, 15]). Interestingly, these models actually implicitly assume a reliable channel between honest voters and the voting server: whenever a voter votes, she first sends her vote to the voting server on a secure channel, before letting the attacker see it. This model prevents an attacker from reading and blocking a message, while this can be easily done in practice (by breaking the connection).

The execution times are very similar between the two tools. The corresponding results are reported in Figure 3.4.

Protocols	ProVerif	TypeEq
Helios	x	0.006s
Needham-Schroeder (sym)	0.17s	0.017s
Needham-Schroeder-Lowe	0.055s	0.010s
Yahalom-Lowe	0.43s	0.020s
Private Authentication	0.025s	0.009s
BAC	0.026s	0.006s

Figure 3.4 – Experimental results for an unbounded number of sessions

Conclusion to Part I

We presented a novel type system for verifying trace equivalence in security protocols. It can be applied to a large class of protocols, with support for else branches, standard cryptographic primitives, dynamic key generation, non-uniform branching, as well as bounded and unbounded numbers of sessions.

We believe that our prototype implementation demonstrates that this approach is promising and opens the way to the development of an efficient technique for proving equivalence properties in even larger classes of protocols.

Our type system requires a light type annotation: the user must specify the type of the keys. While it seems reasonable to expect the user to provide the label (is the key supposed to be secret or not), the type of the payload the key should encrypt or sign can be more complex. In our case study, in most cases this type simply describes the structure and security labels of the messages encrypted by the key. Such types could possibly be inferred automatically by looking at the messages the keys encrypt in the process. However the case of the Helios protocol in particular is rather intricate: the user has to write a refined type that corresponds to an overapproximation of any encrypted message. As future work on this topic, it would be interesting to develop an automatic type inference system, and in particular to explore whether such types could be inferred automatically. A possible heuristic to do this would be to start the typechecking without the information on the payload types for the keys, until we reach a point where a key is used to encrypt a message. A type for that message would then be determined, using its structure and the currently available typing environment, and added as a payload type to the key. We would then have to backtrack, and start the typechecking over with this new key type, until we encounter a message that cannot be typechecked with it. Again the type for this message would be added to the key type, we would backtrack, and so on until – hopefully – a key type is reached that lets the whole process typecheck.

We also plan to study how to add phases to our framework, in order to cover more properties, that involve several stages in the protocol. For instance, a variant of the unlinkability property could be formulated by considering the scenario where the attacker first observes and interacts with a session run by Alice, and only after this session has ended can interact with a second session, and try to determine whether that second session is also with Alice, or with another user Bob. Incorporating phases to our type system would require to generalise it, to account for the fact that the type of a key may depend on the phase in which it is used. For instance in the unlinkability case, a key could have a type specifying that, during the first phase, it only encrypts messages from Alice, while in the second phase it can additionally encrypt messages from Bob.

Several other interesting problems remain to be studied. Notably, a limitation of our type system is that it does not address processes with too dissimilar structures. Our type system goes beyond diff-equivalence (where the executions must be the same on both sides, and only the messages themselves may differ), *e.g.* allowing else branches to be matched with then branches.

However we cannot prove equivalence of processes where traces of P are dynamically mapped to traces of Q, depending on the attacker's behaviour. Such cases occur for example when proving unlinkability of the biometric passport. An interesting direction would be to explore how to enrich our type system with additional rules that could cover such cases, taking advantage of the modularity of the type system. Intuitively, this would require us to express for instance that a key is used in a different way, *e.g.* paired with a different key in a bikey, depending on which branch of a conditional we are typechecking.

Finally, another interesting problem would be the treatment of primitives with algebraic properties (*e.g.* Exclusive Or, or homomorphic encryption). A direction to explore would be to extend the type system with simple rules – compared to the current ones for encryption – that discharge these primitives and their difficulty to the consistency of the constraints. Algebraic properties of primitives intuitively seem easier to handle in the constraints, since they capture the static case.

Part II

Vote Privacy

Introduction to Part II

In order to carefully analyse electronic voting systems, several classes of security requirements have been defined. As explained in introduction, two of the main security properties are:

- privacy: no one should know how I voted;
- verifiability, which is typically described through the three following sub-properties.
  - *individual verifiability:* a voter can check that her ballot is counted;
  - *universal verifiability:* anyone can check that the result corresponds to the published ballots;
  - *eligibility verifiability:* only legitimate voters may vote.

These two main properties seem antagonistic. Basically, privacy requires that as little information as possible should be made public by the authorities, besides the result of the election. On the other hand, for a system to be verifiable, some information has to be published, to allow voters to verify – in some way specified by the scheme – that their vote was counted. Intuition hence seems to indicate that making a verifiable system creates a risk that some information will be revealed on the votes, causing privacy breaches.

Related work In fact, as mentioned in introduction, an impossibility result has even been established [49], stating that verifiability is incompatible with unconditional privacy. That notion of privacy requires that the votes remain perfectly secret even from an adversary with unlimited computational power. No such result however has been established when considering privacy notions that feature a polynomially bounded attacker. A polynomial attacker is usual when studying cryptographic protocols – most protocols achieve only such computational privacy, rather than unconditional privacy. Many game-based privacy definitions have been proposed, as surveyed in [26]: they all consider a polynomial adversary.

[70] establishes a hierarchy between privacy, receipt-freeness, and coercion resistance, while in a quantitative setting, [88] shows that this hierarchy does not hold anymore. [58] recasts several definition of verifiability in a common setting, providing a framework to compare them. Besides [49], none of these approaches relates privacy with verifiability.

Contribution – privacy and verifiability In the following chapters, we study the relations between these two security properties. Our first important contribution is to establish that, in fact, (computational) *privacy implies individual verifiability*, that is, guarantees that all the honest votes will be counted. This result holds for arbitrary primitives and voting protocols without anonymous channels. To show that this implication is not due to a choice of a very particular definition, over the next two chapters, we prove it in two very distinct contexts, namely symbolic and cryptographic models (respectively, Chapters 4 and 5). In symbolic models, such as

the one we considered in Part I, messages are represented by terms and the attacker's behaviour is typically axiomatised through a set of logical formulas or rewrite rules. Cryptographic models are more precise. They represent messages as bitstrings and consider attackers that can be any probabilistic polynomial time Turing machines. Proofs of security are made by reduction to well accepted security assumptions such as hardness of factorisation or discrete logarithm. In both models, we consider a standard notion of privacy already used to analyse several protocols. In the symbolic setting, as explained in previous chapters, we express privacy as an equivalence property between two scenarios where the votes of two voters Alice and Bob are swapped. In the computational case, we use (an adaptation of) Benaloh's definition [24]. That game-based definition follows a similar intuition to the symbolic notion. The adversary wins if he can distinguish between two scenarios where the set of votes from all (honest) voters produces the same result – which generalises the symbolic case, where this set contains only Alice and Bob's votes, and is therefore the same in the two scenarios. In both settings, we establish that privacy implies individual verifiability for a (standard) basic notion of individual verifiability, namely that the result of the election must contain the votes of all honest voters.

To briefly explain our idea, let us consider a very simple protocol, not at all verifiable. In this simple protocol, voters simply encrypt their votes with the public key of the election. The ballot box stores the ballots and, at the end of the election, it provides the list of recorded ballots to the talliers, who detain the private key, possibly split in shares. The talliers compute and publish the result of the election. The ballot box is not public and no proof of correct decryption is provided so voters have no control over the correctness of the result. Such a system is of course not satisfactory but it is often viewed as a "basic" system that can be used in contexts where only privacy is a concern. Indeed, it is typically believed that such a system guarantees privacy provided that the attacker does not have access to the private key of the election. In particular, the ballot box (that is, the voting server) seems powerless. This is actually *not* the case. If the ballot box aims at knowing how a particular voter, say Alice, voted, he may simply keep Alice's ballot in the list of recorded ballots and then replace all the other ballots by encryptions of valid votes of his choice, possibly following a plausible distribution, to make the attack undetected. When the result of the election is published, the ballot box already knows the value of all votes but Alice's, and will therefore be able to deduce how Alice voted. We show how to generalise this idea to produce an attack on privacy from any attack on individual verifiability.

Our result may seem counter-intuitive when thinking of existing analysis of voting schemes. Indeed, if privacy implies individual verifiability, how is it possible to prove Helios [28] or Civitas [17] without even modelling the verification aspects? How can a system that is not fully verifiable like the Neuchâtel protocol be proved private [78]? This apparent contradiction can be resolved by closely examining the trust assumptions we consider: we in fact prove that privacy implies individual verifiability *with the same trust assumptions w.r.t.* all election authorities for both properties. In previous work, verifiability is typically studied against a dishonest ballot box, while privacy notions assume a trusted ballot box.

Contribution – privacy against a dishonest ballot box Our second contribution in this part is to propose new notions of vote privacy, in the context of an untrusted ballot box. Indeed, as pointed out above – a similar observation was made in [29] – existing cryptographic definitions of privacy implicitly assume an honest voting ballot box. That is the case for all definitions surveyed in [26]: honest ballots are assumed to be properly stored and then tallied. Actually, we notice that the same situation occurs in symbolic models. Although the well adopted definition of privacy [71] does not specify how the ballot box should be modelled, most symbolic proofs of

privacy (see *e.g.* [71, 66, 65, 17]) actually assume that the votes of honest voters always reach the ballot box without being modified and that they are properly tallied. The reason is that the authors were aware of the fact that if the adversary may block all ballots but Alice's ballot, he can obviously break privacy. However, to avoid this apparently systematic attack, they make a very strong assumption: the ballot box needs to be honest. This means that previous cryptographic and symbolic privacy analyses only hold assuming an honest ballot box while the corresponding voting systems aim at privacy *without trusting the ballot box*. This seriously weakens the security analysis and attacks may be missed, like the attack of Roenne [94] on Helios, for which there is no easy fix.

In Chapter 6, we attempt to close that gap, by proposing a new definition for ballot privacy, that circumvents this problem and remains meaningful in the context of an untrusted ballot box. Intuitively, vote privacy tries to capture the idea that, no matter how voters vote, the attacker should not be able to see any difference. The key issue is that the result of the election *does* leak some information (*e.g.* the sum of the votes) and the adversary may notice a difference based on this. Typically, as soon as some voters do not check (in the way specified by the voting scheme) that their votes are counted, an adversary may remove these votes without being detected, which leads to the result leaking more information than it should on the remaining votes. Our idea is to solve this issue by designing a definition that allows us to acknowledge this inevitable leakage, and specifically express that we do not wish to consider it as an attack on privacy.

We actually go further than that. In some schemes, such as Helios, when voters do not verify, the adversary is even able to not only remove, but also replace their ballots with his own, containing votes of his choice. Similarly to the discussion above, this additional ability may let him obtain even more information on the remaining votes from the tally. Depending on the context and use case of the scheme, this may or may not be considered an issue. Hence we may wish to specify that such a behaviour – modifying votes of people who do not check – does or does not constitute an attack on privacy.

To handle such cases in a general way, we thus formally define a family of games for privacy against a dishonest ballot box, that explicitly exclude some adversarial behaviours – removing votes, changing them, *etc.*– from constituting attacks. We do this in a modular way: our definition is a cryptographic game that is parametrised by the class of behaviours we wish to allow, *i.e.* not to consider an attack. To show that our definition is still sufficiently robust, we relate it to a family of simulation-based notions of privacy: in these, the voting scheme must securely implement an ideal functionality, that describes precisely what power an adversary can have against the scheme. We show that each instance of our game-based notion implies simulation-based security for a corresponding ideal functionality. Finally we conduct a case study, applying our definition to several existing voting schemes, that prevent or allow different classes of adversarial behaviours, to showcase its flexibility.

Related publications Chapter 4 and Chapter 5 are based on the results from the article [64], written with Véronique Cortier and published at the CCS'18 conference (ACM Conference on Computer and Communications Security). Chapter 6 presents yet unpublished work, which was conducted with Véronique Cortier and Bogdan Warinschi.

Chapter 4

Privacy Implies Verifiability: Symbolic Model

In this chapter, we establish that privacy implies individual verifiability for voting systems in a symbolic model. We first present our model for voting protocols and the associated security properties in the symbolic setting. We then detail under which assumptions our result holds, and formally state and prove it.

4.1 Model

Let us first introduce the symbolic model we will use to represent voting systems. In this section, we consider a generalisation of the process algebra presented in Chapter 1, in the spirit of the applied pi-calculus [6].

4.1.1 Messages

The model here is quite similar to the one we considered earlier in Chapter 1, except we now allow any signature of function symbols, to model any cryptographic primitive, rather than the fixed set of constructors and destructors considered before.

As in Part I, we consider an infinite set of names $\mathcal{N}$ that model fresh values such as nonces and keys. We distinguish the set $\mathcal{FN}$ of free nonces (generated by the attacker) and the set $\mathcal{BN}$ of bound nonces (generated by the protocol agents). We also assume an infinite set of variables $\mathcal{V} = \mathcal{X} \uplus \mathcal{AX}$ where $\mathcal{X}$ contains variables used in processes (agent's memory) while $\mathcal{AX}$ contains variables used to store messages (adversary's memory). Cryptographic primitives are represented through a *signature* $\mathcal{F}$, *i.e.* a set of function symbols with their arity. We assume an infinite set $\mathcal{C} \subseteq \mathcal{F}$ of public constants, which are functions of arity 0.

As before, we model messages by terms: the set $\mathcal{T}(\mathcal{F}, \mathcal{V}, \mathcal{N})$ of terms is inductively defined by applying functions to variables in $\mathcal{V}$ and names in $\mathcal{N}$. The set of names (resp. variables) occurring in a term t is denoted $\mathrm{names}(t)$ (resp. $\mathrm{vars}(t)$). A term is *ground* if it does not contain any variable. The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{AX}, \mathcal{FN})$ represents the *attacker terms*, that is, terms built from the messages sent on the network and stored thanks to the variables in $\mathcal{AX}$. As in Part I, we call *substitution* a mapping from variables to messages, and we denote by $t\sigma$ the application of σ to a term t, *i.e.* the term obtained by replacing each variable $x \in \mathrm{dom}(\sigma)$ occurring in t with $\sigma(x)$.

Contrary to the model considered in Part I however, we do not consider a specific signature $\mathcal{F}$ modelling a fixed set of primitives. Instead, we leave the signature abstract, as our result is general with respect to the signature used.

Indeed, depending on the voting protocol we wish to model, the signature may need to contain symbols for cryptographic primitives not covered by the signature considered in Part I. For instance, some protocols use homomorphic encryption, so that the encrypted votes can be added together, and only the aggregated ciphertext needs to be decrypted. Another example would be randomisable encryption. In some protocols it is important that ballots can be re-randomised: ciphertexts are randomised with a random value when created, and given a ciphertext c, anyone can generate a new c' encrypting the same plaintext as c by updating the random value used in c, without knowing the decryption key. This may be used for instance to have the server in the election re-randomise all ballots before publishing them, so that they still contain the same votes but can no longer be linked with voters. One last example would simply be the $+$ operator, *i.e.* an associative and commutative operator, that can be used to sum all votes together to compute the result of the election. Such primitives are not covered by the signature from Chapter 1.

In addition their behaviour cannot always be modelled as a destructor application similar to the ones used in Chapter 1. Instead, the properties of the cryptographic primitives are modelled through an *equational theory*.

Definition 21 (Equational theory)**.** *An* equational theory E*, is a finite set of equations of the form* $M = N$*, where* $M, N \in \mathcal{T}(\mathcal{F}, \mathcal{X}, \emptyset)$ *are messages without names.*

Equality modulo E, denoted by $=_{\mathsf{E}}$, is defined as the smallest equivalence relation on terms that makes M and N equal for each equation $M = N$ in E, and is closed under context and substitution. Formally:

Definition 22 (Equality modulo an equational theory)**.** *Given an equational theory* E*,* $=_{\mathsf{E}}$ *is the smallest equivalence relation on terms such that*

- *for all equation* $M = N$ *in* E*,* $M =_{\mathsf{E}} N$*.*
- *for all function symbol* $f \in \mathcal{F}$ *of arity* n*, for all messages* $M_1, \dots M_n, N_1, \dots N_n$*, if* $\forall i.\ M_i =_{\mathsf{E}} N_i$ *then*
$$f(M_1, \dots, M_n) =_{\mathsf{E}} f(N_1, \dots, N_n).$$
- *for all messages* M*,* N*, for all substitution* σ*, if* $M =_{\mathsf{E}} N$ *then* $M\sigma =_{\mathsf{E}} N\sigma$*.*

We denote disequalities modulo E by $M \neq_{\mathsf{E}} N$.

A consequence of this modelling, and of considering arbitrary signatures and equational theories, is that we no longer have the restriction from Chapter 1 that keys must be atomic.

Example 6. *Consider the signature* $\mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{C}$ *from Chapter 1. Recall that it contained constructors and destructors for symmetric and asymmetric encryption, signatures, and pairs. We can easily define the properties of these primitives using an equational theory, by turning the rewriting rules for destructors (from Section 1.2) into equations, as follows.*

$$\begin{aligned}
\mathtt{dec}(\mathtt{enc}(x, y), y) &= x \\
\mathtt{adec}(\mathtt{aenc}(x, \mathtt{pk}(y)), y) &= x \\
\mathtt{checksign}(\mathtt{sign}(x, y), \mathtt{vk}(y)) &= x \\
\pi_1(\langle x, y\rangle) &= x \\
\pi_2(\langle x, y\rangle) &= y
\end{aligned}$$

Processes:

P, Q	$::=$	
	0	
	$\mathtt{new}\, n.P$	for $n \in \mathcal{BN}$ (n bound in P)
	$\mathtt{out}(c, M).P$	
	$\mathtt{in}(c, x).P$	for $x \in \mathcal{X}$ (x bound in P)
	$\mathsf{event}(M_1, \ldots, M_n).P$	for $\mathsf{event} \in \mathsf{Ev}$ of arity n
	$P \| Q$	
	$\mathtt{let}\ x = M\ \mathtt{in}\ P$	for $x \in \mathcal{X}$ (x bound in P)
	$\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q$	
	$!P$	

where $M, N, M_1, \ldots, M_n$ are messages and $c \in \mathcal{C}h$ is a channel.

Figure 4.1 – Syntax for processes.

We could enrich it with a $+$ symbol representing the sum, and characterise $+$ as an associative and commutative operator using the following equations.

$$x + (y + z) = (x + y) + z$$
$$x + y = y + x$$

Homomorphic encryption could then for instance be formalised by adding a second associative and commutative operator $*$, and the following equation.

$$\mathtt{aenc}(x, z) * \mathtt{aenc}(y, z) = \mathtt{aenc}(x + y, z).$$

Randomisable encryption could be modelled for instance by adding a third argument to the `aenc` *symbol, that represents the randomness used, and a* `rerand` *symbol for the randomisation operation. Randomising a ciphertext that uses randomness r with a new random value r' would be modelled by the equation*

$$\mathtt{rerand}(\mathtt{aenc}(x, r, y), r') = \mathtt{aenc}(x, r + r', y).$$

4.1.2 Processes

The behaviour of protocol parties is described through processes. The syntax of the process calculus is depicted in Figures 4.1 and 4.2. As usual, we identify processes up to α-renaming, to avoid capture of bound names and variables.

As in Chapter 1, the semantics of processes is given through a transition relation $\xrightarrow{\alpha}$ on configurations, given in Figure 1.2, where α is the *action* associated to the transition. We still denote by $\xrightarrow{w}_*$ the reflexive transitive closure of $\xrightarrow{\alpha}$, where w is the concatenation of all actions.

Both the syntax and semantics of the calculus are very similar to those presented in Chapter 1, with a few additions which we will now present.

First, while it was useful in Part I to consider the symbolic, uninstantiated frame and the instantiation of variables, this is no longer needed here. Hence, for better readability, we combine the two together. What we will now call a frame ϕ is a substitution with $\mathrm{dom}(\phi) \subseteq \mathcal{AX}$ that contains ground messages, and represents the instantiated messages sent on the network. It corresponds to what would earlier have been $\psi\sigma$, where ψ was the sequence of uninstantiated messages, and σ the instantiation of variables.

$$(\{P_1||P_2\} \cup \mathcal{P}; \phi) \xrightarrow{\tau} (\{P_1, P_2\} \cup \mathcal{P}; \phi) \quad \textsc{Par}$$

$$(\{0\} \cup \mathcal{P}; \phi) \xrightarrow{\tau} (\mathcal{P}; \phi) \quad \textsc{Zero}$$

$$(\{\mathtt{new}\, n.P\} \cup \mathcal{P}; \phi) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi) \quad \textsc{New}$$

$$(\{\mathtt{out}(c, M).P\} \cup \mathcal{P}; \phi) \xrightarrow{\mathtt{new}\, ax_n.\mathtt{out}(c, ax_n)} (\{P\} \cup \mathcal{P}; \phi \cup \{M/ax_n\}) \quad \textsc{Out}$$
if M is a ground term, $ax_n \in \mathcal{AX}$ and $n = |\phi| + 1$

$$(\{\mathtt{in}(c, x).P\} \cup \mathcal{P}; \phi) \xrightarrow{\mathtt{in}(c, R)} (\{P[R\phi/x]\} \cup \mathcal{P}; \phi) \quad \textsc{In}$$
if R is an attacker term such that $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi)$

$$(\{\mathsf{event}(M_1, \ldots, M_n).P\} \cup \mathcal{P}; \phi) \xrightarrow{\mathsf{event}(M_1, \ldots, M_n)} (\{P\} \cup \mathcal{P}; \phi) \quad \textsc{Event}$$
if $\forall i.\ M_i$ is a ground message

$$(\{\mathtt{let}\ x = M\ \mathtt{in}\ P\} \cup \mathcal{P}; \phi) \xrightarrow{\tau} (\{P[M/x]\} \cup \mathcal{P}; \phi) \quad \textsc{Let-In}$$
if M is ground

$$(\{\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi) \xrightarrow{\tau} (\{P\} \cup \mathcal{P}; \phi) \quad \textsc{If-Then}$$
if M, N are ground messages such that $M =_{\mathsf{E}} N$

$$(\{\mathtt{if}\ M = N\ \mathtt{then}\ P\ \mathtt{else}\ Q\} \cup \mathcal{P}; \phi) \xrightarrow{\tau} (\{Q\} \cup \mathcal{P}; \phi) \quad \textsc{If-Else}$$
if M, N are ground messages such that $M \neq_{\mathsf{E}} N$

$$(\{!P\} \cup \mathcal{P}; \phi) \xrightarrow{\tau} (\{P, !P\} \cup \mathcal{P}; \phi) \quad \textsc{Repl}$$

Figure 4.2 – Semantics

Moreover, since we no longer consider a fixed set of constructor and destructors, but rather an arbitrary signature and equational theory, we no longer restrict the `let` construction to a fixed list of destructors. Instead, any message can be stored in any variable.

In addition, we extend the calculus with different *communication channels*: when sending or receiving messages, processes must now specify a channel name, that identifies which channel is used. All channels will be public: the attacker has access to all of them and can *e.g.* intercept messages from one channel and resend them on another. We consider different channels nevertheless to model the fact that an attacker can identify the provenance of a message. When a message is sent or received, he can now observe on which channel this action is performed, which gives him potentially useful information when trying to distinguish between two processes.

Formally, we consider an infinite set $\mathcal{C}h$ of channel names, representing the channels on which the messages are exchanged. As seen in Figure 4.1, the `out` and `in` instructions now take as an additional argument a channel name. Accordingly, rules Out and In (Figure 4.2) now record in the trace the channel on which the action is performed.

Finally, we also extend the calculus with *events*. These are steps in the processes that do not actually model the behaviour of the protocol, but are rather used to record that participants have reached a certain step in the protocol, with some associated knowledge. Events can be emitted by processes, and can contain some messages the process wants to record. They are stored in the trace, but are invisible to the attacker. That is, in the eyes of the attacker, two traces with the same observable actions (inputs, outputs) are the same, even if they contain different events. Events are a rather standard feature in symbolic models, for instance when modelling agreement properties between two parties. Such cases can be modelled by having both parties emit an event containing the message (*e.g.* a session key) on which they think they agree. Basically, agreement is then modelled by requiring that, for instance, any event emitted by the responder is matched by an event emitted by the initiator containing the same message. As we will explain in the next section, we use events here to record (secretly to the attacker) which

voters have voted, and for which candidate, in order to express verifiability properties.

Formally, we consider a finite set Ev of event symbols, given together with their arity. The syntax of processes (Figure 4.1) allows them to trigger an event $\mathsf{event} \in \mathsf{Ev}$ of arity n with messages $M_1, \ldots, M_n$ by using instruction $\mathsf{event}(M_1, \ldots, M_n)$. As displayed in Figure 4.2, when this instruction is reached in an execution, the event, including the messages $M_1, \ldots, M_n$, is registered in the trace. As we said, events are invisible to the attacker: formally, they are considered similarly as τ for the equality $=_\tau$. That is, two sequences w, w' that only differ by adding or removing silent actions τ or events will be considered equal up to silent actions, written $w =_\tau w'$.

As in Chapter 1, a *trace* of a process P is any possible sequence of transitions starting from P. Traces correspond to all possible executions in the presence of an attacker that may read, forge, and send messages. Formally, the set of traces $\mathrm{trace}(P)$ is defined as follows.

$$\mathrm{trace}(P) = \{(w, \phi) | (\{P\}; \emptyset) \xrightarrow{w}_* (\mathcal{P}; \phi)\}$$

We will in addition consider the following notion of *blocking* trace.

Definition 23 (blocking trace)**.** *A sequence of actions t is* blocking *in a process P if it cannot be executed. Formally:*

$$\mathsf{blocking}(t, P) \stackrel{\text{def}}{=} \forall \phi.\ (t, \phi) \notin \mathrm{trace}(P).$$

4.1.3 Equivalence

We consider the same notion of trace equivalence as in Part I, rewritten here with the equational theory-based formalism. Two sequences of messages are indistinguishable to an attacker if he cannot perform any test that could distinguish them. Two processes P and Q are in equivalence if no matter how the adversary interacts with P, a similar interaction may happen with Q, with equivalent resulting frames.

Definition 24 (Static Equivalence)**.** *Two ground frames ϕ and ϕ' are statically equivalent if and only if they have the same domain, and for all attacker terms R, S with variables in $\mathrm{dom}(\phi) = \mathrm{dom}(\phi')$, we have*

$$(R\phi =_{\mathsf{E}} S\phi) \iff (R\phi' =_{\mathsf{E}} S\phi')$$

Definition 25 (Trace Equivalence)**.** *Let P, Q be two processes. We write $P \sqsubseteq_t Q$ if for all $(s, \phi) \in \mathrm{trace}(P)$, there exists $(s', \phi') \in \mathrm{trace}(Q)$ such that $s =_\tau s'$ and ϕ and ϕ' are statically equivalent. We say that P and Q are trace equivalent, and we write $P \approx_t Q$, if $P \sqsubseteq_t Q$ and $Q \sqsubseteq_t P$.*

4.2 Voting Systems and Security Properties

In this section, we present the modelling of voting protocols as pi-calculus processes we will use, as well as the formal definitions of the security properties we consider in this framework.

4.2.1 Notations

We start with a few notations. The multiset of elements a, a, b, c is denoted $\{\!|a, a, b, c|\!\}$. The union of two multisets S_1 and S_2 is denoted $S_1 \uplus S_2$. If $k \in \mathbb{N}$, for any value v, $k \cdot v$ denotes the

multiset containing k instances of v. If S is a multiset, we denote $S(v)$ the number of instances of v in S.

We assume a set V of messages representing possible votes (*e.g.* constants) and a set $\mathcal{R}$ of values representing possible results. We also assume $\mathcal{R}$ is equipped with an associative and commutative operator $*$. This operator maps two elements of $\mathcal{R}$ onto an element of $\mathcal{R}$, and represents the operation of combining results together (*e.g.* adding them). This operator does not have to be a function symbol from the signature $\mathcal{F}$, as illustrated in the following example.

Example 7. *For instance, with the signature and equational theory from Example 6, + would be a valid choice for $*$, if $\mathcal{R}$ is* e.g. *the set of all messages.*

If $\mathcal{R}$ is the set of pairs of messages, another example of a $*$ operator would be the function

$$\langle M, N\rangle, \langle M', N'\rangle \mapsto \langle M + M', N + N'\rangle.$$

The values in $\mathcal{R}$ do not necessarily have to be messages: they can be abstract values not defined in terms of the signature and equational theory considered. For instance, $\mathcal{R}$ could contain multisets of votes (*i.e.* multisets of messages). In our symbolic model, the result of the election $r \in \mathcal{R}$ will then be encoded as a term. Depending on the choice of this encoding, the same abstract result $r \in \mathcal{R}$ may have several *representations*, *i.e.* several terms may encode the same abstract result. For instance, in the case of multisets, say the signature does not contain a construction for multisets, but only for lists of terms. Then a multiset may be represented *e.g.* by any list containing its elements, in any order. We want our result to be independent of a particular choice of representation. Therefore, we will simply assume the existence of a *representation function*, that associates a result to messages representing it:

Definition 26 (Representation)**.** *A* representation R *is a function that associates to an abstract result $r \in \mathcal{R}$ a finite set of possible representations,* i.e. *a finite set of messages that encode this result, with an injectivity property:*

$$\forall r \neq r'.\ R(r) \cap R(r') = \emptyset$$

Intuitively, a result can be associated to several representations, but a given representation must be unambiguous, i.e. *it can correspond to at most one result.*

A *counting function* is a function ρ that associates a result $r \in \mathcal{R}$ to a multiset of votes. We assume that counting functions have a *partial tally* property: it is always possible to count the votes in two distinct multisets and then combine the results.

Definition 27 (Counting function with partial tallying)**.** *A counting function ρ has* partial tallying *if*

$$\forall V, V'\ \rho(V \uplus V') = \rho(V) * \rho(V')$$

A vote is said to be *neutral* if it does not contribute to the result.

Definition 28 (Neutral vote)**.** *A vote v is* neutral w.r.t. *a counting function ρ if $\rho(v)$ is neutral* w.r.t. $*$.

Example 8. *Consider a finite set of constants representing candidates $\mathcal{C} = \{a_1, \ldots, a_k\}$. In case voters should select between k_1 and k_2 candidates or vote blank, assuming a construction for*

vectors in the signature $\mathcal{F}$ considered, we can represent valid votes by vectors that encode the selection of candidates

$$\mathsf{V}_{k_1,k_2} = \left\{ v \in \{0,1\}^k \;\middle|\; k_1 \leq \sum_{i=0}^{k} v_i \leq k_2 \right\} \cup \{v^{\mathsf{blank}}\}$$

where v^{blank} is the null vector $(0,\ldots,0)$, representing a blank vote. For a vote $v \in \mathsf{V}_{k_1,k_2}$, v_i being 1 means that candidate a_i has been selected, and v_i being 0 means it has not.

In a voting scheme with mixnet-based tally, all the individual votes are shuffled before being revealed, in a random order. Thus the set $\mathcal{R}$ of results is the set of multisets of votes in V_{k_1,k_2} and $*$ is the union of multisets. The corresponding counting function is $\rho_{mix}(V) = V$, where V is a multiset of elements of V_{k_1,k_2}.

In a system with homomorphic-based tally, the votes are added together. Thus $\mathcal{R} = \mathbb{N}^k$ is the set of vectors of k elements, and $*$ is the addition of vectors. The corresponding counting function is $\rho_{hom}(V) = \sum_{v \in V} v$.

Both ρ_{mix} and ρ_{hom} have the partial tally property. The vote v^{blank} is a neutral vote w.r.t. *ρ_{hom} but not ρ_{mix}. Indeed adding a null vector does not change the sum of the votes, but it does change the multiset of votes.*

4.2.2 Voting protocols

We consider two disjoint, infinite subsets of $\mathcal{C}$: a set $\mathcal{A}$ of *agent names* or *identities*, and a set V of *votes*. We assume given a representation function R of the result.

A voting protocol is modelled as a process. It is composed of:

- processes that represent honest voters;
- a process modelling the tally;
- possibly other processes, modelling other authorities.

Formally, we define a *voting process* as follows.

Definition 29 (Voting process)**.** *A* voting process *is a process of the form*

$$\begin{aligned} P = \mathtt{new}\ \overrightarrow{cred}.\mathtt{new}\ cred_1 \ldots \mathtt{new}\ cred_p.\ (& \\ & \mathsf{Voter}(a_1, v_{a_1}, \vec{c_1}, \overrightarrow{cred}, cred_1) \mid \cdots \mid \mathsf{Voter}(a_n, v_{a_n}, \vec{c_n}, \overrightarrow{cred}, cred_n) \\ & \mid \mathsf{Tally}_p(\vec{c}, \overrightarrow{cred}, cred_1, \ldots, cred_p) \\ & \mid \mathsf{Others}_p(\vec{c}', \overrightarrow{cred}, cred_1, \ldots, cred_p)) \end{aligned}$$

*where n is the number of honest voters, $a_i \in \mathcal{A}$, $v_{a_i} \in \mathsf{V}$, $\vec{c_i}$, $\vec{c}$, $\vec{c}'$ are (distinct) channels, $\overrightarrow{cred}$ and $cred_i$ are (distinct) names. $\overrightarrow{cred}$ represents the election credentials, that all voters have access to (*e.g. *a public key to encrypt their votes). $cred_i$ represents the private credential that voter i has access to (*e.g. *a signature key).*

A voting process may be instantiated with various voters and vote selections. Given $A = \{b_1,\ldots,b_n\} \subseteq \mathcal{A}$ a finite set of voters, and a mapping $\alpha : A \to \mathcal{V}$ that associates a vote to each voter, we define P_α by replacing a_i with b_i and v_i with $\alpha(b_i)$ in P. We call such a mapping α a distribution of votes.

Moreover, P must satisfy the following properties.

- Process $\mathsf{Voter}(a, v_a, \vec{c}, \overrightarrow{cred}, cred)$ models an honest voter a willing to vote for v_a, using the channels $\vec{c}$. credentials $cred$ (*e.g.* a signing key) and election credentials $\overrightarrow{cred}$. It is assumed to contain an event $\mathsf{Voted}(a, v)$ that models that a has voted for v. This event is typically placed at the end of process $\mathsf{Voter}(a, v_a, \vec{c}, \overrightarrow{cred}, cred)$. This event cannot appear in process Tally_p nor Others_p.

- Process $\mathsf{Tally}_p(\vec{c}, \overrightarrow{cred}, cred_1, \ldots, cred_p)$ models the tally. It is parametrised by the total number of voters p (honest and dishonest), with $p \geq n$. It is assumed to contain exactly one output action on a reserved channel c_r. The term output on this channel is assumed to represent the final result of the election, in any execution trace. Formally, we require that this term is at least always the representation of some result:

$$\forall \alpha.\ \forall (tr, \phi) \in \mathrm{trace}(P_\alpha).\ \mathtt{out}(c_r, r) \in tr \ \Rightarrow\ \exists V.\ \phi(r) \in R(\rho(V))$$

  Tally_p may of course contain other input/output actions, on other channels.

- Process $\mathsf{Others}_p(\vec{c}', \overrightarrow{cred}, cred_1, \ldots, cred_p)$ is an arbitrary process, also parametrised by p. It models the remainder of the voting protocol, for example the behaviour of other authorities. It also models the initial knowledge of the attacker by sending appropriate data (*e.g.* the public key of the election or dishonest credentials). To be more generic, we leave this process unspecified, and simply assume that it uses a set of channels disjoint from the channels used in Voter and Tally_p.

The channel c_r used in Tally_p to publish the result is called the *result channel* of P.

Note that this notion of voting process does not assume honest authorities, *e.g.* a honest tally. We only require that the tally is modelled by some process, but this process does not necessarily follow the specification of the voting protocol. For instance, to model a fully dishonest tally controlled by the attacker, process Tally_p could simply output its entire knowledge, including the election private key k_e, to the attacker, and receive instructions from the attacker (on some channel other than c_r) regarding which result to publish on c_r.

Example 9. *The models of the Helios protocol presented earlier (*e.g. *Sections 1.3 or 3.2) can easily be adapted into a model that fits this generic notion of voting process.*

For clarity, we describe here a simple version with only two honest voters A and B, a dishonest voter C, and a voting server S. The protocol can be modelled by the following process.

$$\begin{array}{l} P_{\mathsf{Helios}}(v_a, v_b) = \\ \quad \mathtt{new}\, k_a, k_b, k_c, k_e. \\ \qquad (\mathtt{out}(c, k_c).\mathtt{out}(c, \mathtt{pk}(k_e)) \ \mid \\ \qquad \mathsf{Voter}(A, v_a, c_a, c'_a, k_a, k_e) \ \mid\ \mathsf{Voter}(B, v_b, c_b, c'_b, k_b, k_e) \ \mid \\ \qquad \mathsf{Tally}_{\mathsf{Helios}}(c_a, c_b, c_c, c_s, k_a, k_b, k_c, k_e)) \end{array}$$

$\mathsf{Voter}(a, v, c, c', k, k_e)$ *simply sends vote v encrypted with the election public key $\mathtt{pk}(k_e)$. To model the fact that voters communicate with the ballot box through an authenticated channel, we assume, as before, that a voter Alice sends her ballot to the server signed with a signature key k, which we will consider as her credential. Note that this is just a modelling artefact to*

abstract away the underlying password-based authenticated channel. After sending her ballot, Alice triggers the Voted *event to indicate she has voted.*

$$\mathsf{Voter}(a, v, c, c', k, k_e) = \mathtt{new}\, r.\ \mathtt{out}(c, \mathtt{sign}(\mathtt{aenc}(\langle v, r\rangle, \mathtt{pk}(k_e)), k)).\ \mathsf{Voted}(a, v).$$

The voting server receives ballots from voters A, B, and C, performs weeding as explained earlier, and then outputs the decrypted ballots, after some mixing, which is modelled through the $+$ *operator from Example 6.*

$$\begin{array}{l}\mathsf{Tally}_{\mathsf{Helios}}(c_a, c_b, c_c, c_s, k_a, k_b, k_c, k_e) = \\ \quad \mathtt{in}(c_a, x_1).\mathtt{in}(c_b, x_2).\mathtt{in}(c_c, x_3). \\ \quad \mathtt{let}\ y_1 = \mathtt{checksign}(x_1, \mathtt{vk}(k_a))\ \mathtt{in} \\ \quad \mathtt{let}\ y_2 = \mathtt{dec}(x_2, \mathtt{vk}(k_b))\ \mathtt{in} \\ \quad \mathtt{let}\ y_3 = \mathtt{dec}(x_3, \mathtt{vk}(k_c))\ \mathtt{in} \\ \quad \mathtt{if}\ x_1 \neq x_2\ \wedge\ x_1 \neq x_3\ \wedge\ x_2 \neq x_3\ \mathtt{then} \\ \qquad \mathtt{out}(c_s, \pi_1(\mathtt{adec}(y_1, k_e))\ +\ \pi_1(\mathtt{adec}(y_2, k_e))\ +\ \pi_1(\mathtt{adec}(y_3, k_e)))\end{array}$$

where we omit the null else-branches. $\wedge$ *is syntactic sugar for a succession of tests, and* $\mathtt{if}\ M \neq N\ \mathtt{then}\ P$ *is syntactic sugar for* $\mathtt{if}\ M = N\ \mathtt{then}\ 0\ \mathtt{else}\ P$.

Here, the process Others_p *consists in the output of the election public key, and the credential of the dishonest voter C controlled by the attacker:* $\mathtt{out}(c, k_c).\mathtt{out}(c, \mathtt{pk}(k_e))$.

We can use the Voted event to read which voters voted from a trace. Formally:

Definition 30 (Voters in a trace)**.** *Given a sequence tr of actions, the set of voters* $\mathsf{Voters}(tr)$ *who did vote in tr is defined as follows.*

$$\mathsf{Voters}(tr) = \{a \in \mathcal{A}\ \mid\ \exists v \in \mathcal{V}.\ \mathsf{Voted}(a, v) \in tr\}.$$

The result of the election is emitted on a special channel c_r, and can thus also be read from a trace. Formally:

Definition 31. *Given a trace* (t, ϕ) *and a multiset of votes V, the predicate* $\mathsf{result}(t, \phi, V)$ *holds if the election result in* (t, ϕ) *corresponds to V. That is, if t ends with an output on the result channel* c_r*, and* ϕ *indicates that this output is a valid representation of* $\rho(V)$*:*

$$\mathsf{result}(t, \phi, V)\ =\ \exists x, t'.\ t = t'.\mathtt{out}(c_r, x)\ \wedge\ \phi(x) \in R(\rho(V)).$$

4.2.3 Security properties

We now formally define in our model the two properties we study: individual verifiability and vote privacy.

Verifiability. Several definitions of verifiability have been proposed. In the lines of [87, 57], we consider a very basic notion, that says that the result should at least contain the votes from honest voters.

Definition 32 (Symbolic individual verifiability)**.** *Let P be a voting process with result channel c_r. P satisfies* symbolic individual verifiability *if, for any distribution α of votes, and any trace $(t, \phi) \in \text{trace}(P_\alpha)$ that ends with an output on c_r, there exists a multiset of votes V_c such that the result in t corresponds to $V_a \uplus V_c$, where V_a is the multiset of honest votes.*

That is, P is individually verifiable if

$$\forall \alpha.\ \forall (t, \phi) \in \text{trace}(P_\alpha).\ V(t, \phi)$$

where

$$V(t, \phi) \quad = \quad \forall t', x.\ (t = t'.\mathtt{out}(c_r, x)) \ \Rightarrow\ \exists V_c.\ \phi(x) \in R(\rho(\{\!|v \mid \exists a.\ \mathsf{Voted}(a, v) \in t|\!\} \uplus V_c)).$$

This property intuitively means that whenever a result is produced in an execution of the protocol, this result must account for at least all honest votes produced in this execution (plus some arbitrary set of dishonest ones).

Usually, individual verifiability typically guarantees that voters can check that their ballot will be counted. Our notion of individual verifiability goes one step further, requiring that the corresponding votes will appear in the result, even if the tally is dishonest. Note that this definition of verifiability does not explicitly mention any verification steps performed by voters. Typically, voting schemes specify some steps voters should take to ensure their vote is counted, such as for instance checking that their encrypted ballot correctly appears on a public bulletin board published by the authorities. Our notion does not explicitly deal with these steps, but requires instead that a vote is counted as soon as the Voted event is triggered. In some cases (depending on which protocol is modelled, and on whether the authorities are assumed honest in the model considered), this may only be achievable if the voter process correctly performs the verification steps before triggering the Voted event.

One of the first definitions of verifiability was given in [86], distinguishing between individual, universal, and eligibility verifiability. Intuitively, our own notion of individual verifiability sits somewhere between individual verifiability and individual plus universal verifiability as defined in [86]. A precise comparison is difficult as individual and universal verifiability are strongly tied together in [86]. Moreover, [86] only considers the case where all voters are honest and they all vote.

Privacy. We consider the privacy definition already presented in Section 2.2, initially proposed in [71] and widely adopted in symbolic models: an attacker cannot distinguish when Alice is voting v_1 and Bob is voting v_2 from the scenario where the two votes are swapped.

Definition 33 (Privacy)**.** *Let P be a voting process. P satisfies* privacy *if, for any distribution of votes α, for any two voters $a, b \in \mathcal{A} \backslash \text{dom}(\alpha)$ and any two votes $v_1, v_2 \in \mathcal{V}$, we have*

$$P_{\alpha \cup \{a \mapsto v_1, b \mapsto v_2\}} \approx P_{\alpha \cup \{a \mapsto v_2, b \mapsto v_1\}}$$

Having formally defined the properties we consider, we can now state and prove our main result.

4.3 Main Result

We show that privacy implies verifiability under a couple of assumptions, typically satisfied in practice.

4.3.1 Assumptions

First, we assume a light form of determinacy: two traces with the same observable actions yield the same election result. This excludes for example cases for voters choose non deterministically how they vote. Formally, we define the notion of *election determinacy.*

Definition 34 (Election determinacy)**.** *We say that a voting process P with result channel c_r is* election determinate *if, for any distribution of votes α, for any two sequences of actions t, t' such that $t =_\tau t'$, if $(t.\mathtt{out}(c_r, x), \phi) \in \mathrm{trace}(P_\alpha)$, and $(t'.\mathtt{out}(c_r, x), \phi') \in \mathrm{trace}(P_\alpha)$, then*

$$\forall V.\ \phi(x) \in R(\rho(V))) \Rightarrow \phi'(x) \in R(\rho(V))$$

This assumption still supports some form of non determinism but may not hold for example in the case where voters use anonymous channels that even hide who participated in the election.

Second, we assume that it is always possible for a new voter to vote (before the tally started) without modifying the behaviour of the protocol. We formalise this assumption as the notion of *voting friendliness.*

Definition 35 (Voting friendliness)**.** *A voting process P is* voting friendly *if for all voter $a \in \mathcal{A}$, there exists t'' (the honest voting trace) such that for all α satisfying $a \notin \mathrm{dom}(\alpha)$,*

- *for all $(t, \phi) \in \mathrm{trace}(P_\alpha)$, such that $t = t'.\mathtt{out}(c_r, x)$ for some t', x, for all v, there exists tr, ψ such that $tr =_\tau t''$, $\mathsf{Voted}(a, v) \in tr$, $(t'.tr.\mathtt{out}(c_r, x), \psi) \in \mathrm{trace}(P_{\alpha \cup \{a \mapsto v\}})$, and $\forall V.\ \phi(x) \in R(\rho(V)) \Rightarrow \psi(x) \in R(\rho(V \cup \{v\}))$. Intuitively, if a votes normally, her vote will be counted as expected, no matter how the adversary interfered with the other voters.*
- *for all t', x such that $\mathsf{blocking}(t'.\mathtt{out}(c_r, x), P_\alpha)$, for all v, tr, ψ such that $tr =_\tau t''$, we have $\mathsf{blocking}(t'.tr.\mathtt{out}(c_r, x),\ P_{\alpha \cup \{a \mapsto v\}})$. Intuitively, the fact that a voted does not suddenly unlock the tally if it was blocked before.*

In practice, most voting systems are voting friendly since voters vote independently. In particular, process P_{Helios} modelling Helios, as defined in Example 2, is voting friendly. The voting friendly property prevents a fully dishonest tally since the first item requires that unmodified honest ballots are correctly counted. However, we can still consider a partially dishonest tally that, for example, discards or modifies ballots that have been flagged by the attacker.

Finally, we need an assumption that some vote is counted in a particular way. The reason why such an assumption is useful will become clear when proving our theorem. Formally, we identify two variants of this condition. Our result holds provided one of the two following assumptions is true:

1. There exists a neutral vote $v_{\mathsf{neutral}} \in \mathcal{V}$, such that $\rho(\{\!|v_{\mathsf{neutral}}|\!\})$ is neutral for $*$.
2. There exists a special vote $v_{\mathsf{special}} \in \mathcal{V}$, which is counted separately in the result. That is, v_{special} must satisfy the following properties.

   Definition 36 (Special vote)**.** *A vote v_{special} is said to be special if*

   - *the result associated with a multiset determines the number of instances of v_{special}:*

   $$\forall V, V'.\ \rho(V) = \rho(V') \implies V(v_{\mathsf{special}}) = V'(v_{\mathsf{special}}).$$

- *and the count of* v_{special} *can be simplified*

$$\forall V, V', k.\ \rho(V \uplus k \cdot v_{\mathsf{special}}) = \rho(V' \uplus k \cdot v_{\mathsf{special}}) \implies \rho(V) = \rho(V').$$

For example, for ρ_{hom}, all the votes are special, therefore this property always holds. For ρ_{mix}, it depends on the set of valid votes. In the standard case where a vote is a selection of candidates (for example between k_1 and k_2 candidates), then a special vote could be, for instance, a vote that includes the selection of an extra candidate, not used before.

4.3.2 Theorem

We can now state and prove our main result about symbolic privacy and verifiability.

Theorem 8 (Privacy implies individual verifiability)**.** *Let P be a voting friendly, election determinate voting process. Assuming there exists either a neutral vote or a special vote, if P satisfies privacy then P satisfies individual verifiability.*

The proof of this result intuitively relies on the fact that in order to satisfy privacy *w.r.t.* two voters Alice and Bob, a voting process has to guarantee that the vote of Alice is, if not correctly counted, at least taken into account to some extent. For instance, if an attacker, trying to distinguish whether Alice voted for 0 and Bob for 1, or Alice voted for 1 and Bob for 0, is able to make the tally ignore completely the vote of Alice, the result of the election is then Bob's choice. Hence the attacker learns how Bob voted, which breaks privacy.

Therefore, we first prove that if a protocol satisfies privacy, then if we compare an execution (*i.e.* a trace) where Alice votes 0 with the corresponding execution where Alice votes 1, the resulting election results must differ by exactly a vote for 0 and a vote for 1. Formally, we show the following lemma.

Lemma 19 (Privacy implies F)**.** *If a voting friendly, election determinate voting process P satisfies privacy, then it satisfies the following property F:*

$$\begin{aligned} F \ = \forall \alpha.\ \forall a \in \mathcal{A} \backslash \mathrm{dom}(\alpha).\ \forall v_1, v_2 \in \mathcal{V}.\ \forall t, t', \phi, \phi', V, V'. \\ [t =_\tau t' \ \wedge\ (t, \phi) \in \mathrm{trace}(P_{\alpha \cup \{a \mapsto v_1\}}) \ \wedge\ (t', \phi') \in \mathrm{trace}(P_{\alpha \cup \{a \mapsto v_2\}}) \wedge \\ \quad \mathsf{result}(t, \phi, V) \ \wedge\ \mathsf{result}(t', \phi', V')\,] \quad \implies \\ \rho(V' \uplus \{\!|v_1|\!\}) = \rho(V \uplus \{\!|v_2|\!\}). \end{aligned}$$

Proof. We prove this by contradiction: assuming F does not hold, we construct an attack on privacy.

Assume F is false. Hence there exists a scenario where changing the vote of one agent does not change the result by one. That is to say, there exist a distribution of votes α, an agent $a \notin \mathrm{dom}(\alpha)$, votes $v_1, v_2 \in \mathcal{V}$, traces $(t, \phi) \in \mathrm{trace}(P_{\alpha \cup \{a \mapsto v_1\}})$ and $(t', \phi') \in \mathrm{trace}(P_{\alpha \cup \{a \mapsto v_2\}})$ such that $t =_\tau t'$, and two multisets V, V', such that $\mathsf{result}(t, \phi, V)$, $\mathsf{result}(t', \phi', V')$, and $\rho(V' \uplus \{\!|v_1|\!\}) \neq \rho(V \uplus \{\!|v_2|\!\})$.

Since $\mathsf{result}(t, \phi, V)$, there exist x, t_1 such that $t = t_1.\mathtt{out}(c_r, x)$ and $\phi(x) \in R(\rho(V))$. Similarly, there exist y, t_1' such that $t' = t_1'.\mathtt{out}(c_r, y)$ and $\phi'(y) \in R(\rho(V'))$. Since $t =_\tau t'$, $x = y$.

Note that we necessarily have $v_1 \neq v_2$: indeed, assume $v_1 = v_2$. Then (t, ϕ) and (t, ϕ') are traces of the same process $P_{\alpha \cup \{a \mapsto v_1\}} = P_{\alpha \cup \{a \mapsto v_2\}}$. Since $\phi(x) \in R(\rho(V))$, by the election determinacy assumption (Definition 34), this implies that $\phi'(x) \in R(\rho(V))$. Since we already

know that $\phi'(x) \in R(\rho(V'))$, and since by assumption R is injective (Definition 26), we have $\rho(V) = \rho(V')$. Thus, as $v_1 = v_2$, we have $\rho(V) * \rho(\{\!|v_2|\!\}) = \rho(V') * \rho(\{\!|v_1|\!\})$, which is contradictory. Hence $v_1 \neq v_2$.

The attack on privacy consists in the fact that, since changing a's vote does not produce a change of exactly one in the result, even in presence of another agent b whose vote is the opposite of a's, the result will be different depending on the vote of a.

Formally, let $b \notin \mathrm{dom}(\alpha) \cup \{a\}$. By the voting friendliness assumption (Definition 35), b can cast a vote after the traces t and t'. That is, there exist sequences of actions t_b, t'_b, and frames ψ, ψ', such that

$$(t_1.t_b.\mathtt{out}(c_r, x), \psi) \in \mathrm{trace}(P_{\alpha\cup\{a\mapsto v_1, b\mapsto v_2\}}),$$
$$(t'_1.t'_b.\mathtt{out}(c_r, x), \psi') \in \mathrm{trace}(P_{\alpha\cup\{a\mapsto v_2, b\mapsto v_1\}}),$$
$$\mathsf{Voted}(b, v_2) \in t_b, \quad \mathsf{Voted}(b, v_1) \in t'_b, \quad t_b =_\tau t'_b,$$
$$\psi(x) \in R(\rho(V \uplus \{\!|v_2|\!\})), \text{ and } \psi'(x) \in R(\rho(V' \uplus \{\!|v_1|\!\})).$$

Since $\rho(V' \uplus \{\!|v_1|\!\}) \neq \rho(V \uplus \{\!|v_2|\!\})$, by the injectivity assumption on R, we have $\psi(x) \neq \psi'(x)$.

We have constructed two frames, obtained by the same actions in $P_{\alpha\cup\{a\mapsto v_1, b\mapsto v_2\}}$ and $P_{\alpha\cup\{a\mapsto v_2, b\mapsto v_1\}}$, which yield different results for the election. Using the election determinacy assumption, this lets us prove that these two processes are not trace equivalent.

Indeed, let us denote $t_2 \stackrel{\text{def}}{=} t_1.t_b.\mathtt{out}(c_r, x)$ and $t'_2 \stackrel{\text{def}}{=} t'_1.t'_b.\mathtt{out}(c_r, x)$. We have $(t_2, \psi) \in \mathrm{trace}(P_{\alpha\cup\{a\mapsto v_1, b\mapsto v_2\}})$. For any trace $(t''_2, \psi'') \in \mathrm{trace}(P_{\alpha\cup\{a\mapsto v_2, b\mapsto v_1\}})$, if $t''_2 =_\tau t_2 =_\tau t'_2$, then by election determinacy we have $\psi''(x) \in R(\rho(V' \uplus \{\!|v_1|\!\}))$. Hence, by injectivity of R, since $\rho(V \uplus \{\!|v_2|\!\}) \neq \rho(V' \uplus \{\!|v_1|\!\})$, we have $\psi''(x) \notin R(\rho(V \uplus \{\!|v_2|\!\}))$. Checking whether a message belongs to $R(\rho(V \uplus \{\!|v_2|\!\}))$ is a test the attacker can perform, as this set is finite by definition of R. Since $\psi(x) \in R(\rho(V \uplus \{\!|v_2|\!\}))$, ψ, ψ'' are therefore not statically equivalent.

Thus, the trace (t_2, ψ) of $P_{\alpha\cup\{a\mapsto v_1, b\mapsto v_2\}}$ cannot be mimicked by any trace of $P_{\alpha\cup\{a\mapsto v_2, b\mapsto v_1\}}$. In other words, $P_{\alpha\cup\{a\mapsto v_1, b\mapsto v_2\}} \not\approx_t P_{\alpha\cup\{a\mapsto v_2, b\mapsto v_1\}}$. This violates P, which concludes the proof. □

This lemma is used as a central property to prove the theorem. Intuitively, the rest of the proof will be as follows. We apply this lemma repeatedly, changing one by one all the votes from honest voters into neutral or special votes (depending on which one we assume). Let r denote the result before this operation, and r' the result after. Let V_a denote the multiset of honest votes, and V_b the multiset containing the same number of neutral (or special) votes. Thanks to Lemma 19, we can show that $r * \rho(V_b) = r' * \rho(V_a)$. The properties of neutral and special votes will let us prove that this implies r contains all votes in V_a. The neutral case is perhaps the clearest: then V_b only contains neutral votes, and we have $r = r' * \rho(V_a)$. This means that r contains all honest votes, hence the voting process satisfies individual verifiability.

Formally, we show the following lemma, that uses the property F from Lemma 19. It describes how the result changes when we turn any subset V_{wanted} of the votes cast by honest voters into any arbitrary multiset of votes V_{change} (with the same cardinality). Following the intuition exposed earlier, we will later on use the set of all honest votes V_a as V_{wanted}, and neutral or special votes as V_{change}.

Lemma 20 (Privacy and F imply FF)**.** *Consider a voting process P that is election determinate and voting friendly. Assume P satisfies privacy, and the F property from Lemma 19. Then P satisfies the following property FF:*

$$\begin{aligned} FF \stackrel{\text{def}}{=} \forall \alpha.\, \forall (t,\phi) \in \mathrm{trace}(P_\alpha).\, \forall V, V_{\mathsf{change}}.\, \forall V_{\mathsf{wanted}} \subseteq \{\!|v \mid \exists a.\, \mathsf{Voted}(a,v) \in t|\!\}. \\ \mathsf{result}(t,\phi,V) \;\Rightarrow \\ |V_{\mathsf{change}}| = |V_{\mathsf{wanted}}| \;\Rightarrow \\ \exists V_c.\, \rho(V) * \rho(V_{\mathsf{change}}) = \rho(V_{\mathsf{wanted}}) * \rho(V_c) \end{aligned}$$

Proof. Assume that both privacy and F hold. Consider a distribution α of votes, and a trace $(t,\phi) \in \mathrm{trace}(P_\alpha)$. Let V be such that $\mathsf{result}(t,\phi,V)$, *i.e.* there exist t', x such that $t = t'.\mathtt{out}(c_r, x)$ and $\phi(x) \in R(\rho(V))$. Let $V_{\mathsf{wanted}} \subseteq \{\!|v \mid \exists a.\, \mathsf{Voted}(a,v) \in t|\!\}$, and V_{change} such that $|V_{\mathsf{change}}| = |V_{\mathsf{wanted}}|$.

To prove FF, we need to show that the result in this trace augmented with V_{change} contains at least the subset V_{wanted} of the (intended) votes of the honest voters. That is to say, we must show that there exists V_c such that $\rho(V) * \rho(V_{\mathsf{change}}) = \rho(V_{\mathsf{wanted}}) * \rho(V_c)$.

The idea of the proof is to compare $\rho(V)$ to the result $\rho(V')$ obtained by turning, one by one, all votes from V_{wanted} into the votes from V_{change}, and performing the same sequence of actions. As we will see, this is possible, otherwise privacy would break; and $V \uplus V_{\mathsf{change}}$ contains more instances of the honest votes than $V' \uplus V_{\mathsf{wanted}}$, since F holds.

Let us denote the Voted events appearing in t (not necessarily in that order) by

$$\mathsf{Voted}(a_1, v_1), \ldots, \mathsf{Voted}(a_l, v_l)$$

for some pairwise distinct agents $a_1, \ldots, a_l \in \mathsf{Voters}(t)$, and some $l \in \mathbb{N}$.

By definition, each element of V_{wanted} is associated with one of these Voted events. Let $m \stackrel{\text{def}}{=} |V_{\mathsf{wanted}}|$. Without loss of generality, we may assume that $V_{\mathsf{wanted}} = \{\!|v_1, \ldots, v_m|\!\}$.

Note that $|V_{\mathsf{change}}|$ is also equal to m by assumption. Let us then denote $V_{\mathsf{change}} \stackrel{\text{def}}{=} \{\!|v'_1, \ldots, v'_m|\!\}$.

Since, by assumption on the form of the processes, the $\mathsf{Voted}(a,v)$ event can only be emitted by the process $\mathsf{Voter}(a,v,c)$ for some credential c, we have $\alpha(a_i) = v_i$ for all $i \in [\![1,m]\!]$.

For $i \in [\![0,m]\!]$, let α_i denote the affectation of votes obtained from α by turning the first i votes from V_{wanted} to V_{change}, *i.e.*

- $\alpha_i(a_j) = v'_j$ if $j \in [\![1,i]\!]$;
- $\alpha_i(a_j) = v_j$ if $j \in [\![i+1,m]\!]$;
- $\alpha_i(a) = \alpha(a)$ if $a \in \mathrm{dom}(\alpha)$ is not one of the a_j, $j \in [\![1,m]\!]$.

Let $\beta \stackrel{\text{def}}{=} \alpha_m$. Note that $\alpha_0 = \alpha$, and that all the α_i have the same domain.

Let us show that for all i, the same actions as t can be performed in P_{α_i} with the same agents emitting Voted events, *i.e.* that

$$\forall i \in [\![0,m]\!].\, \exists t_i.\, t_i =_\tau t \;\wedge\; \neg \mathsf{blocking}(t_i, P_{\alpha_i}).$$

By contradiction, assume this property does not hold, and let i be the smallest index that falsifies it. Hence,

$$\forall t_i.\, t_i =_\tau t \;\Rightarrow\; \mathsf{blocking}(t_i, P_{\alpha_i}). \tag{4.1}$$

In addition, note that since the property is clearly satisfied at the 0th step, $i \geq 1$. Hence, it holds for $i-1$, *i.e.* there exists t_{i-1} such that $t_{i-1} =_\tau t$, and $\neg\mathsf{blocking}(t_{i-1}, P_{\alpha_{i-1}})$. Since $t = t'.\mathtt{out}(c_r, x)$, we also have $t_{i-1} = t'_{i-1}.\mathtt{out}(c_r, x)$ for some t'_{i-1} such that $t'_{i-1} =_\tau t'$.

Then, for all $t'_i =_\tau t'_{i-1}$ $(=_\tau t')$, by (4.1), $\mathsf{blocking}(t'_i.\mathtt{out}(c_r, x), P_{\alpha_i})$ holds.

The same sequence of actions t_{i-1} is blocking at step i and not at step $i-1$, which differs only by the vote of of a_i. This lets us construct an attack on privacy, which constitutes a contradiction. Indeed, by the voting friendliness assumption, we may add a voter $b \notin \mathrm{dom}(\alpha)$, who votes for v'_i at step $i-1$, and for v_i at step i, and there exists tr such that

- there exists $tr' =_\tau tr$ and ψ such that $(t'_{i-1}.tr'.\mathtt{out}(c_r, x), \psi) \in \mathrm{trace}(P_{\alpha_{i-1} \cup \{b \mapsto v'_i\}})$.
- for all $t'_i =_\tau t'_{i-1}$ $(=_\tau t')$, we have shown that $\mathsf{blocking}(t'_i.\mathtt{out}(c_r, x), P_{\alpha_i})$, and thus for all $tr'' =_\tau tr$ and all ψ', $(t'_i.tr''.\mathtt{out}(c_r, x), \psi') \notin \mathrm{trace}(P_{\alpha_i \cup \{b \mapsto v_i\}})$.

Therefore, the processes $P_{\alpha_i \cup \{b \mapsto v_i\}}$ and $P_{\alpha_{i-1} \cup \{b \mapsto v'_i\}}$ are not trace equivalent. Since they only differ by the votes of a_i and b, who respectively vote for v_i, v'_i on the left and v'_i, v_i on the right, this breaks privacy, which contradicts the hypotheses.

Thus, for all i, there exists t_i such that $t_i =_\tau t$, of the form $t'_i.\mathtt{out}(c_r, x)$, such that $\neg\mathsf{blocking}(t_i, P_{\alpha_i})$. In other words, there exists ϕ_i such that $(t_i, \phi_i) \in \mathrm{trace}(P_{\alpha_i})$. By assumption, only results may get published on c_r. Hence, there exists V_i such that $\phi_i(x) \in R(\rho(V_i))$, *i.e.* $\mathsf{result}(t_i, \phi_i, V_i)$. Let $V' \stackrel{\text{def}}{=} V_m$. Note that $V_0 = V$.

For all $i \in [\![0, m-1]\!]$, α_i and α_{i+1} only differ by the vote of a_{i+1}, which is v'_{i+1} in α_{i+1} and v_{i+1} in α_i. Hence, by F, we have $\rho(V_i \uplus \{\!|v'_{i+1}|\!\}) = \rho(V_{i+1} \uplus \{\!|v_{i+1}|\!\})$.

That is to say, since ρ is assumed to have partial tallying, that

$$\rho(V_i) * \rho(\{\!|v'_{i+1}|\!\}) = \rho(V_{i+1}) * \rho(\{\!|v_{i+1}|\!\}).$$

Therefore, by rewriting these m equalities successively, we have

$$\rho(V_0) * \rho(\{\!|v'_1|\!\}) * \cdots * \rho(\{\!|v'_m|\!\}) = \rho(V_m) * \rho(\{\!|v_1|\!\}) * \cdots * \rho(\{\!|v_m|\!\}),$$

i.e., again by partial tallying,

$$\rho(V) * \rho(\{\!|v'_i \mid i \in [\![1, m]\!]|\!\}) = \rho(\{\!|v_i \mid i \in [\![1, m]\!]|\!\}) * \rho(V').$$

By definition, this means

$$\rho(V) * \rho(V_{\mathsf{change}}) = \rho(V_{\mathsf{wanted}}) * \rho(V')$$

which concludes the proof. □

As announced earlier, we now apply property *FF* to either neutral or special votes, and use it to establish individual verifiability.

First, let us show the case of the neutral vote.

Lemma 21 (*FF* implies V with a neutral vote)**.** *Consider a voting process P that is election determinate and voting friendly. Assume that there exists a neutral vote v_{neutral}. If P satisfies property FF (from Lemma 20), then P satisfies individual verifiability.*

Proof. Let α be a distribution of votes, and let $(t'.\mathtt{out}(c_r, x), \phi) \in \mathrm{trace}(P_\alpha)$. Let $t \stackrel{\mathrm{def}}{=} t'.\mathtt{out}(c_r, x)$. To prove individual verifiability, we need to show that the result in this trace contains at least the (intended) votes of the honest voters. That is to say, we must show that there exists V_c such that $\phi(x) \in R(\rho(\{\!|v \mid \exists a.\, \mathsf{Voted}(a, v) \in t|\!\} \uplus V_c))$.

By assumption, only results are published on channel c_r. Hence there exists some multiset of votes V such that $\phi(x) \in R(\rho(V))$. We then have $\mathsf{result}(t, \phi, V)$.

Let $V_{\mathsf{wanted}} \stackrel{\mathrm{def}}{=} \{\!|v \mid \exists a.\, \mathsf{Voted}(a, v) \in t|\!\}$ be the multiset of all intended honest votes in t. Let $k \stackrel{\mathrm{def}}{=} |V_{\mathsf{wanted}}|$, and $V_{\mathsf{change}} \stackrel{\mathrm{def}}{=} k \cdot v_{\mathsf{neutral}}$.

By *FF*, there exists a multiset V_c such that $\rho(V) * \rho(V_{\mathsf{change}}) = \rho(V_{\mathsf{wanted}}) * \rho(V_c)$.

Since ρ is assumed to have partial tallying, $\rho(V_{\mathsf{change}}) = \rho(\{\!|v_{\mathsf{neutral}}|\!\})^k$. As v_{neutral} is a neutral vote, $\rho(\{\!|v_{\mathsf{neutral}}|\!\})$ is neutral for $*$, and thus so is $\rho(V_{\mathsf{change}})$.

Therefore, $\rho(V) = \rho(V_{\mathsf{wanted}}) * \rho(V_c) = \rho(V_{\mathsf{wanted}} \uplus V_c)$, which proves the claim. □

The case of a special vote is slightly more involved.

Lemma 22 (*FF* implies V with a special vote)**.** *Consider a voting process P that is election determinate and voting friendly. Assume that there exists a special vote v_{special}. If P satisfies property FF (from Lemma 20), then P satisfies individual verifiability.*

Proof. Let α be a distribution of votes, and let $(t'.\mathtt{out}(c_r, x), \phi) \in \mathrm{trace}(P_\alpha)$. Let $t \stackrel{\mathrm{def}}{=} t'.\mathtt{out}(c_r, x)$. To prove individual verifiability, we need to show that the result in this trace contains at least the (intended) votes of the honest voters. That is to say, we must show that there exists V_c such that $\phi(x) \in R(\rho(\{\!|v \mid \exists a.\, \mathsf{Voted}(a, v) \in t|\!\} \uplus V_c))$.

By assumption, only results are published on channel c_r. Hence there exists V such that $\phi(x) \in R(\rho(V))$. We then have $\mathsf{result}(t, \phi, V)$.

Let $V_{\mathsf{wanted}} \stackrel{\mathrm{def}}{=} \{\!|v \mid \exists a.\, \mathsf{Voted}(a, v) \in t \ \wedge\ v \neq v_{\mathsf{special}}|\!\}$ be the multiset of all intended honest votes in t that are not v_{special}.

Let $k \stackrel{\mathrm{def}}{=} |V_{\mathsf{wanted}}|$, and $V_{\mathsf{change}} \stackrel{\mathrm{def}}{=} k \cdot v_{\mathsf{special}}$. By *FF*, there exists a multiset V_c such that

$$\rho(V) * \rho(V_{\mathsf{change}}) = \rho(V_{\mathsf{wanted}}) * \rho(V_c).$$

We may rewrite this equality as

$$\rho(V \uplus k \cdot v_{\mathsf{special}}) = \rho(V_{\mathsf{wanted}} \uplus V_c)$$

by the partial tallying property of ρ. Since v_{special} is assumed to be special (Definition 36), we can deduce that

$$(V \uplus k \cdot v_{\mathsf{special}})(v_{\mathsf{special}}) = (V_{\mathsf{wanted}} \uplus V_c)(v_{\mathsf{special}}).$$

Yet, $V_{\mathsf{wanted}}(v_{\mathsf{special}}) = 0$ by definition. Therefore, $V_c(v_{\mathsf{special}}) \geq k$, and we may decompose V_c into $V_c = V_c' \uplus k \cdot v_{\mathsf{special}}$ for some V_c'. We then have

$$\rho(V \uplus k \cdot v_{\mathsf{special}}) = \rho(V_{\mathsf{wanted}} \uplus V_c' \uplus k \cdot v_{\mathsf{special}}),$$

which implies, by definition of a special vote, that

$$\rho(V) = \rho(V_{\mathsf{wanted}} \uplus V_c').$$

Since V_{wanted} contains all the intended honest votes different from v_{special}, it remains to be proved that V'_c contains sufficiently many instances of v_{special}.

Let $k' \stackrel{\text{def}}{=} |\{\!|\mathsf{Voted}(a, v_{\mathsf{special}}) \in t \mid a \in \mathcal{A}\}\!|$ the number of intended votes for v_{special} in t; and $V'_{\mathsf{wanted}} \stackrel{\text{def}}{=} k' \cdot v_{\mathsf{special}}$.

Let $v \in \mathcal{V}$ such that $v \neq v_{\mathsf{special}}$. Let $V'_{\mathsf{change}} \stackrel{\text{def}}{=} k' \cdot v$.

By *FF*, there exists V''_c such that

$$\rho(V) * \rho(V'_{\mathsf{change}}) = \rho(V'_{\mathsf{wanted}}) * \rho(V''_c),$$

i.e., by partial tallying,

$$\rho(V \uplus k' \cdot v) = \rho(k' \cdot v_{\mathsf{special}} \uplus V''_c).$$

By definition of a special vote, we then have

$$(V \uplus k' \cdot v)(v_{\mathsf{special}}) = (k' \cdot v_{\mathsf{special}} \uplus V''_c)(v_{\mathsf{special}}).$$

As before, since $v \neq v_{\mathsf{special}}$, this means that $V(v_{\mathsf{special}}) \geq k'$.

We already know that $\rho(V) = \rho(V_{\mathsf{wanted}} \uplus V'_c)$. Again by definition of a special vote, we thus have $(V_{\mathsf{wanted}} \uplus V'_c)(v_{\mathsf{special}}) = V(v_{\mathsf{special}}) \geq k'$. Since $v_{\mathsf{special}} \notin V_{\mathsf{wanted}}$, $(V'_c)(v_{\mathsf{special}}) \geq k'$, *i.e.* $V'_c = V'''_c \uplus k' \cdot v_{\mathsf{special}}$ for some V'''_c.

Therefore we have

$$\rho(V) = \rho(V_{\mathsf{wanted}} \uplus k' \cdot v_{\mathsf{special}} \uplus V'''_c) = \rho(\{\!|v \mid \exists a.\, \mathsf{Voted}(a, v) \in t\}\!| \uplus V'''_c),$$

which concludes the proof. □

Theorem 8 then directly follows from Lemmas 19, 20, 21 and 22.

We have established that privacy implies individual verifiability under some mild assumptions on the voting scheme in our symbolic model. At this point, it is interesting to note that this result holds only when considering the same trust assumptions with respect to the election authorities (ballot box, tally, ...) for both properties. That is, privacy against some attacker model implies verifiability against *the same* attacker model. Indeed, in this symbolic model, these trust assumptions are encoded in the way the voting process is written. For instance, a honest ballot box would be modelled by a process that correctly authenticates voters and stores their ballots, while a dishonest one could be modelled by giving control of this step to the attacker, and letting him propose an arbitrary list of ballots to be tallied. This would *e.g.* allow him to remove some ballots after they have been cast. Looking carefully at the statement of Theorem 8, the *same* voting process is considered for privacy and verifiability. Therefore, the encoding of the trust assumptions has to be the same for both. We will discuss this observation later on. For now, in the next chapter, we show that the same implication also holds in a computational model for cryptography.

Chapter 5

Privacy Implies Verifiability: Computational Model

In this chapter, we show that a private voting system must be individually verifiable, in a computational setting. We first describe our model for voting system, and the definitions we consider for privacy and verifiability. In a nutshell, we consider the well established definition of privacy by Benaloh [24], and a notion of individual verifiability which states that the result must account for at least all the votes cast by honest voters. We then detail a few assumptions under which our result holds, and finally prove it.

5.1 Voting Systems and Security Properties

In this section we introduce the model and terminology we consider for voting systems and security properties.

Computational models define protocols and adversaries as probabilistic polynomial-time algorithms.

5.1.1 Voting systems

Let us start with the terminology and notations we will use to describe voting systems. Votes are simply bitstrings. A *ballot* is also a bitstring, that is intended to contain a voter's vote. The exact nature of the ballots depend on the protocol we consider: for instance, they could simply be ciphertexts, or also contain some public credential identifying the voter.

During an election, ballots are typically collected by an authority, and displayed on a *bulletin board*.

Definition 37 (Bulletin board)**.**
A bulletin board BB *is a list of ballots.* $\mathsf{BB}[j]$ *denotes the* j*th element of* BB*.*

We will also in the following use the term *ballot box* interchangeably with *bulletin board*.

As in the symbolic case presented in Chapter 4, we will call *counting function* a function ρ that associates a bitstring to a multiset of votes. That bitstring is intended to represent the result of the election. As before, we will in this Chapter consider counting functions with the partial tallying property, *i.e.* such that $\forall V, V'\ \rho(V \uplus V') = \rho(V) * \rho(V')$, where $*$ is an associative and commutative operation.

In addition, in our cryptographic setting, we will also assume that given an election result r and a multiset of votes V, one can decide efficiently (in polynomial time) whether r includes all the votes of V. That is, we will assume a polynomial-time algorithm algorithm D that decides whether there exists V' such that $r = \rho(V \uplus V')$:

$$\forall r, V. \quad \mathsf{D}(r, V) = 1 \iff \exists V'.\, r = \rho(V \uplus V').$$

This condition is satisfied by the ρ_{mix} and ρ_{hom} functions from Example 8, and all standard counting functions.

Notation: We will in the following consider lists containing pairs (id, x) of a voter's identity and some element x (votes, ballots,...). We may for such a list L write $(id, *) \in \mathsf{L}$ as a shorthand, meaning that there exists an element of the form (id, x) in L for some x. If V is a multiset of elements of the form (id, v), we define $\rho(V) = \rho(\{\!|v \mid (id, v) \in V|\!\})$.

We model voting systems as follows.

Definition 38. *A* voting scheme *consists in seven probabilistic polynomial-time algorithms*

$$\mathcal{V} = (\mathsf{Setup}, \mathsf{Register}, \mathsf{Pub}, \mathsf{Vote}, \mathsf{Tally}, \mathsf{Valid}, \mathsf{ValidTally})$$

where

- $\mathsf{Setup}(1^\lambda)$*, given a security parameter* λ*, computes a pair of* election keys $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Register}(1^\lambda, id)$ *creates a* private credential c *for voter* id*, for example a signing key. The credentials may be empty as well, if the protocol does not use any. The correspondence* (id, c) *is stored in a list* U*, used for modelling purposes. For better readability, we will sometimes refer to this list as a table, and write* $\mathsf{U}[id] = c$ *to denote that* U *contains a correspondence between* id *and* c*,* $\mathsf{U}[id] \leftarrow c$ *to denote that* (id, c) *is added to* U*, and* $\mathsf{U}[id]$ *to denote to the credential associated to* id *in* U.
- $\mathsf{Pub}(c)$ *returns the public credential associated with a credential* c*. For instance, this could be the associated verification key if* c *is a signing key. Again, depending on the protocol, this public credential might be empty.*
- $\mathsf{Vote}(\mathsf{pk}, id, c, v)$ *constructs a ballot containing the vote* v *for voter* id *with credential* c*, using the election public key* pk.
- $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U})$ *uses the board* BB*, the election secret key* sk *and the list of voter identities and credentials* U *to compute the* tally *of the ballots in* BB*,* i.e. *the result of the election, and potentially proofs of correct tallying. The* Tally *algorithm first runs some test* $\mathsf{ValidTally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U})$ *that typically checks that all ballots of* BB *are valid.* Tally *may return* $\perp$ *if the tally procedure fails (invalid board or decryption failure for example). If* $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U}) \neq \perp$ *then it must correspond to a valid result, that is, there exists* V *such that* $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U}) = \rho(V)$.
- $\mathsf{Valid}(id, b, \mathsf{BB}, \mathsf{U}, \mathsf{pk})$ *checks that a ballot* b *cast by a voter* id *is valid with respect to the board* BB *using the election public key* pk*. The nature of this check depends on the protocol considered. For example, the ballot* b *may be required to have a valid signature or valid proofs of knowledge. The ballot* b *will be added to* BB *only if* $\mathsf{Valid}(id, b, \mathsf{BB}, \mathsf{U}, \mathsf{pk})$ *succeeds.*

We will always assume a *consistent* voting scheme, that is, such that tallying honestly generated ballots yields the expected result.

Definition 39 (Consistency)**.** *A voting scheme is* consistent *if for all distinct voter identities* $id_1, \ldots, id_n$*, for all votes* $v_1, \ldots, v_n$*, for all election keys* $(\mathsf{pk}, \mathsf{sk})$ *produced by* Setup*, if for all* i $c_i = \mathsf{Register}(1^\lambda, id_i)$ *and* $\mathsf{U} = [(id_i, c_i) | i \in [\![1, n]\!]]$*, and*

$$\mathsf{BB} = [\mathsf{Vote}(\mathsf{pk}, id_i, c_i, v_i) | i \in [\![1, n]\!]],$$

then with overwhelming probability

$$\mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U}) = \rho(\{\!|v_1, \ldots, v_n|\!\}).$$

The Tally algorithm typically applies a revote policy. When voters may vote several times, the revote policy states which vote should be counted. The two main standard revote policies are 1. the last vote counts or 2. the first vote counts (typically when revote is forbidden). In what follows, our definitions are written assuming the last ballot revote policy. However, they can easily be adapted to the first ballot revote policy and all our results hold in both cases.

To apply a revote policy, the tallying authority must be able to associate each ballot to the voter who cast it. Hence, we will assume a function $\mathsf{extract}_{id}$ which, given a ballot b, the election secret key sk, and the table U of registered voters and credentials, retrieves the identity associated with b. Formally, for any id, c, pk, v, with overwhelming probability

$$\mathsf{extract}_{id}(\mathtt{vote}(\mathsf{pk}, id, c, v), \mathsf{sk}, \mathsf{U}) = id$$

Depending on the scheme, this might be done directly if ballots contain the identity, *e.g.* if the encrypted votes are displayed to the bulletin board next to the corresponding voter's identity. In other cases, ballots may only contain the voter's public credential, *e.g.* their verification key, or in some cases the private credential in an encrypted form. In such cases, the list U could be used to retrieve the identity associated with a credential. Note that we do not require that the tallying authority actually has access to the table U containing the correspondence between identity and credentials, nor that it actually computes $\mathsf{extract}_{id}$. We simply require that such a function exists.

Note that some schemes do not satisfy this condition, in particular when the ballots contain no identifier. Such schemes typically assume that voters do not revote since there is no means to identify whether two ballots originate from the same voter.

Finally, we make three mild assumptions on the tallying and validity check algorithms. The first one is that the tally only counts ballots cast with registered identities, *i.e.*

$$\forall \mathsf{BB}, \mathsf{sk}, \mathsf{U}.\ \mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U}) = \mathsf{Tally}(\mathsf{BB}', \mathsf{sk}, \mathsf{U})$$

where $\mathsf{BB}' = [b \in \mathsf{BB} \mid (\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U}), *) \in \mathsf{U}]$. The second is that registering more voters does not change the tally: if U, U' have no id in common and $\forall b \in \mathsf{BB}.\ (\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U} \parallel \mathsf{U}'), *) \notin \mathsf{U}'$, then

$$\mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U}) = \mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U} \cup \mathsf{U}').$$

The last one is that a ballot being declared valid for some user only depends on that user's credential and the one contained in the ballot, and not of other users' credentials, *i.e.* that for all U, U', id

$$\begin{aligned}(\mathsf{U}[id] = \mathsf{U}'[id] \ \wedge\ \mathsf{U}[\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U})] = \mathsf{U}'[\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U})]) \implies \\ \mathsf{Valid}(id, b, \mathsf{BB}, \mathsf{pk}, \mathsf{U}) = \mathsf{Valid}(id, b, \mathsf{BB}, \mathsf{pk}, \mathsf{U}').\end{aligned}$$

$\mathcal{O}_{\mathsf{reg}}(id)$

if $(id, *) \in \mathsf{U}$ **then stop**
else
  $c \leftarrow \mathsf{Register}(1^\lambda, id)$
  $\mathsf{U}[id] \leftarrow c$
  return $\mathsf{Pub}(\mathsf{U}[id])$

$\mathcal{O}_{\mathsf{corr}}(id)$

if $(id, *) \notin \mathsf{U} \ \vee \ id \in \mathsf{D}$ **then stop**
else
  $\mathsf{D} \leftarrow \mathsf{D} \parallel id$
  return $\mathsf{U}[id]$

Figure 5.1 – Registration and corruption oracles

5.1.2 Security properties

As usual in computational models, an *adversary* is any probabilistic polynomial time Turing machine (PPTM). We define verifiability and privacy through game-based properties.

Verifiability

For verifiability, we propose a simple definition, inspired from [87, 57]. Intuitively, we require that the election result contains at least the votes of all honest voters. This notion was called weak verifiability in [57] but we will call it individual verifiability to match the terminology used in symbolic settings (Chapter 4). More sophisticated and demanding definitions have been proposed, for example controlling how many dishonest votes can be inserted [57] or tolerating some variations in the result [87].

The main missing part (in terms of security) is that our definition does not control ballot stuffing: arbitrarily many dishonest votes may be added to the result. The reason is that ballot stuffing seems unrelated to privacy. Moreover, our definition assumes an honest tally, and thus does not capture universal verifiability aspects. The main reason is that existing privacy definitions in computational settings assume an honest tally, and we compare the two notions under the same trust assumptions.

Verifiability is defined through the game $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda)$ displayed in Figure 5.2. In a first step[14], the adversary may use oracles $\mathcal{O}_{\mathsf{reg}}(id)$ and $\mathcal{O}_{\mathsf{corr}}(id)$ (defined in Figure 5.1) to respectively register a voter and get her credential (in this case, the voter is said to be corrupted, or dishonest). Private credentials are stored in list U, and public credentials are published by $\mathcal{O}_{\mathsf{reg}}$. $\mathcal{O}_{\mathsf{corr}}$ stores the list of corrupted voter identities in D. Then the adversary may ask an honest voter id to vote for a given vote v through oracle $\mathcal{O}^v_{\mathsf{vote}}(id, v)$. In this case, the adversary sees the corresponding ballot. The fact that id voted for v is registered in the list Voted, replacing id's previous votes if necessary, to apply the revote policy. The adversary may also cast an arbitrary ballot b in the name of a dishonest voter id through oracle $\mathcal{O}_{\mathsf{cast}}(id, b)$. Provided b is valid *w.r.t.* BB, it is added to the ballot box.

Finally, once the adversary gives control back to the challenger, the election result is computed. The adversary wins if the result does not contain all the honest votes registered in Voted.

Definition 40 (Individual verifiability)**.** *A voting system is* individually verifiable *if for any adversary* $\mathcal{A}$*,*

$$\mathbb{P}\left[\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda) = 1\right] \text{ is negligible.}$$

[14] In the games, we leave implicit that when the adversary proceeds in several steps, it may transmit a state containing some knowledge to be used in further steps.

$\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},(\mathcal{A}_1,\mathcal{A}_2)}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
$\mathsf{U},\mathsf{D} \leftarrow []$
$\mathcal{A}_1^{\mathcal{O}_{\mathsf{reg}},\mathcal{O}_{\mathsf{corr}}}(\mathsf{pk})$
$\mathsf{BB},\mathsf{Voted} \leftarrow []$
$\mathcal{A}_2^{\mathcal{O}^v_{\mathsf{vote}},\mathcal{O}_{\mathsf{cast}}}(\mathsf{pk})$
$r \leftarrow \mathsf{Tally}(\mathsf{BB},\mathsf{sk},\mathsf{U})$
if $r \neq \perp \ \wedge\ \forall V_c$ (finite) .
  $r \neq \rho(\{v_i\}_{1\leq i\leq k} \uplus V_c)$ **then**
  return 1
where $\mathsf{Voted} = \{(id_1,v_1),\ldots,(id_k,v_k)\}$

$\mathcal{O}^v_{\mathsf{vote}}(id,v)$

if $(id,*) \in \mathsf{U} \wedge id \notin \mathsf{D}$ **then**
  $b \leftarrow \mathsf{Vote}(\mathsf{pk},id,\mathsf{U}[id],v)$
  $\mathsf{BB} \leftarrow \mathsf{BB}\|b$
  $\mathsf{Voted} \leftarrow \mathsf{Voted}'\|(id,v)$
  return b
    where Voted' is obtained from Voted by removing all previous instances of $(id,*)$

$\mathcal{O}_{\mathsf{cast}}(id,b)$

if $id \in \mathsf{D} \wedge$
  $\mathsf{Valid}(id,b,\mathsf{BB},\mathsf{U},\mathsf{pk})$
then
  $\mathsf{BB} \leftarrow \mathsf{BB}\|b$

Figure 5.2 – Verifiability

As mentioned in introduction of Part II, [49] shows an impossibility result between (unconditional) privacy and verifiability. [49] considers another aspect of verifiability, namely universal verifiability, that is, the guarantee that the result corresponds to the content of the ballot, even in presence of a dishonest tally. Interestingly, the same incompatibility result holds between individual verifiability and unconditional privacy, for the same reasons. Exactly like in [49], a powerful adversary (*i.e.* not polynomial) could tally BB and BB' where BB' is the ballot box from which Alice's ballot has been removed and infer Alice's vote by difference. More generally, unconditional privacy is lost as soon as there exists a function of the votes that is meaningfully related to the result, which is implied by individual verifiability.

Privacy

For privacy, we consider the old, well established definition of Josh Benaloh [24], that we slightly extend and adapt to fit our model and notations. More sophisticated definitions are been proposed later. Notably, [26] surveys existing game-based notions of vote privacy, and propose and a unifying definition, which we present and extend in the next chapter. These aim in particular at getting rid of the partial tally assumption (needed in [24]). Note however that they all assume an honest ballot box. Since we also assume partial tally, the original Benaloh definition is sufficient for our needs.

Intuitively, a voting system is private if, no matter how honest voters vote, the adversary cannot see any difference. The game for privacy will formalise this intuition by letting the adversary propose two possible choices of votes for each honest voter. One will be used, and the adversary will try to guess which. However, the adversary always sees the election result, that leaks how the group of honest voters voted (altogether). Therefore, we can only require that the adversary cannot guess which of the two vote choices was used provided that the election result *w.r.t.* the honest voters remains the same for both choices. More formally, in a first step, the adversary uses oracles $\mathcal{O}_{\mathsf{reg}}(id)$ and $\mathcal{O}_{\mathsf{corr}}(id)$ (Figure 5.1) to respectively register a voter and corrupt one to get her credential. As before, in that case, the voter is then said to be corrupted.

The adversary may then request an honest voter id to vote either for v_0 or v_1 through oracle $\mathcal{O}^p_{\mathsf{vote}}(id,v_0,v_1)$. Voter id will vote v_β depending on the bit β. The adversary may also cast an arbitrary ballot b in the name of a dishonest voter id through oracle $\mathcal{O}_{\mathsf{cast}}(id,b)$. The election will be tallied and shown to the adversary only if the set V_0 of votes v_0 yields the same result

$\mathsf{Exp}^{\mathsf{priv},\beta}_{\mathcal{V},\mathcal{A}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
$\mathsf{U},\mathsf{D} \leftarrow []$
$\mathcal{A}^{\mathcal{O}_{\mathsf{reg}},\mathcal{O}_{\mathsf{corr}}}(\mathsf{pk})$
$\mathsf{BB},\mathsf{V}_0,\mathsf{V}_1 \leftarrow []$
$\mathcal{A}^{\mathcal{O}^p_{\mathsf{vote}},\mathcal{O}_{\mathsf{cast}}}(\mathsf{pk})$
if $\rho(\mathsf{V}_0) = \rho(\mathsf{V}_1)$ **then**
  $r \leftarrow \mathsf{Tally}(\mathsf{BB},\mathsf{sk},\mathsf{U})$
  $\beta' \leftarrow \mathcal{A}(\mathsf{pk},r)$
  return β'

$\mathcal{O}^p_{\mathsf{vote}}(id,v_0,v_1)$

if $(id,*) \in \mathsf{U} \ \wedge \ id \notin \mathsf{D}$ **then**
  $b \leftarrow \mathsf{Vote}(\mathsf{pk},id,\mathsf{U}[id],v_\beta)$
  $\mathsf{BB} \leftarrow \mathsf{BB}\|b$
  $\mathsf{V}_0 \leftarrow \mathsf{V}'_0\|(id,v_0)$
  $\mathsf{V}_1 \leftarrow \mathsf{V}'_1\|(id,v_1)$
  return b
    where V'_0 (resp. V'_1) is obtained from V_0 (resp. V_1) by removing all instances of $(id,*)$

$\mathcal{O}_{\mathsf{cast}}(id,b)$

if $id \in \mathsf{D} \wedge$
  $\mathsf{Valid}(id,b,\mathsf{BB},\mathsf{U},\mathsf{pk})$
then
  $\mathsf{BB} \leftarrow \mathsf{BB}\|b$

Figure 5.3 – Privacy

than the set V_1 of votes v_1 (where only the last vote is counted). Finally, the adversary wins if he correctly guesses β. Formally, privacy is defined through the game $\mathsf{Exp}^{\mathsf{priv},\beta}_{\mathcal{V},\mathcal{A}}(\lambda)$ displayed on Figure 5.3.

Definition 41 (Privacy [24])**.** *A voting system is private if for any adversary $\mathcal{A}$,*

$$\left|\mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{V},\mathcal{A}}(\lambda)=1\right]-\mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},1}_{\mathcal{V},\mathcal{A}}(\lambda)=1\right]\right| \text{ is negligible.}$$

Note that this definition of vote privacy implicitly assumes that voters do not communicate with the ballot box through an anonymous channel: indeed, the voting oracle lets the adversary know which voter produced which ballot.

5.2 Main Theorem

We show that privacy implies individual verifiability, under a few assumptions.

5.2.1 Assumptions

As for the symbolic case, we assume the existence of a neutral vote v_{neutral}, that does not contribute to the result (Definition 28).

We also require that the tallying of ballots can be performed piecewise, which is a similar requirement to the partial tallying property of the counting function on votes. That is, informally, as soon as two boards $\mathsf{BB}_1,\mathsf{BB}_2$ are independent then $\mathsf{Tally}(\mathsf{BB}_1 \parallel \mathsf{BB}_2) = \mathsf{Tally}(\mathsf{BB}_1) * \mathsf{Tally}(\mathsf{BB}_2)$. This property is satisfied by most voting schemes. Typically it holds as soon as the tallying algorithm correctly implements a counting function that has the partial tallying operation. Formally, we characterise this notion of "independence" using the $\mathsf{extract}_{id}$ function defined in the previous section.

Definition 42 (Piecewise tally)**.** *A voting scheme has the* piecewise tally property *if for any two boards BB_1 and BB_2 that contain ballots registered for different agents and such that $\mathsf{BB}_1 \parallel \mathsf{BB}_2$ is valid, that is, for all key sk generated by Setup and list of users U created by $\mathsf{Register}$, if*

$$\mathsf{ValidTally}(\mathsf{BB}_1 \parallel \mathsf{BB}_2,\mathsf{sk},\mathsf{U}) \ \wedge \ \forall b \in \mathsf{BB}_1.\ \forall b' \in \mathsf{BB}_2.\ \mathsf{extract}_{id}(b,\mathsf{sk},\mathsf{U}) \neq \mathsf{extract}_{id}(b',\mathsf{sk},\mathsf{U}),$$

$\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V},\mathcal{A}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
$\mathsf{U},\mathsf{D} \leftarrow []$
$\mathcal{A}^{\mathcal{O}_{\mathsf{reg}},\mathcal{O}_{\mathsf{corr}}}(\mathsf{pk})$
$(\mathsf{BB}, id, v) \leftarrow \mathcal{A}^{\mathcal{O}^{vt}_{\mathsf{vote}}}(\mathsf{pk})$
if $(id,*) \notin \mathsf{U}$ **then stop**;
$b \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$
if $id \notin \mathsf{D} \wedge$
$(\forall b' \in \mathsf{BB}.\ \mathsf{extract}_{id}(b', \mathsf{sk}, \mathsf{U}) \neq id) \wedge$
$\mathsf{ValidTally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U}) \wedge$
$\neg\mathsf{ValidTally}(\mathsf{BB}||b, \mathsf{sk}, \mathsf{U})$ **then**
return 1

$\mathcal{O}^{vt}_{\mathsf{vote}}(id, v)$

if $(id,*) \in \mathsf{U} \ \wedge \ id \notin \mathsf{D}$ **then**
return $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$

Figure 5.4 – ValidTally game

then their tally can be computed separately, i.e. *with overwhelming probability:*

$$\mathsf{Tally}(\mathsf{BB}_1 \parallel \mathsf{BB}_2, \mathsf{sk}, \mathsf{U}) = \mathsf{Tally}(\mathsf{BB}_1, \mathsf{sk}, \mathsf{U}) * \mathsf{Tally}(\mathsf{BB}_2, \mathsf{sk}, \mathsf{U}). \tag{5.1}$$

We say that a voting scheme is *strongly correct* if whatever valid board the adversary may produce, adding a honestly generated ballot still yields a valid board. A similar assumption was introduced in [26]. For example, Helios is strongly correct.

Definition 43 (Strong correctness)**.** *A voting scheme $\mathcal{V}$ is* strongly correct *if for any adversary $\mathcal{A}$,*

$$\mathbb{P}\left[\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V},\mathcal{A}}(\lambda) = 1\right] \ \textit{is negligible}$$

where the game $\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V},\mathcal{A}}(\lambda)$ is defined in Figure 5.4.

5.2.2 Theorem

We can now state and prove our main result about computational privacy and verifiability.

Theorem 9 (Privacy implies individual verifiability)**.** *Let $\mathcal{V}$ be a consistent and strongly correct voting scheme that has the piecewise tally property. If $\mathcal{V}$ is private, then $\mathcal{V}$ is individually verifiable.*

Let us first give the general intuition of the proof of this theorem. This proof is inspired by the same intuition as in the symbolic case from Chapter 4. If an attacker manages to break verifiability, that is, to obtain that not all votes from the honest voters are counted correctly, then there also exists an attack against privacy. Indeed, consider a scenario with additional, new voters, whose votes should compensate those cast by the initial voters. By performing the attack on verifiability for the initial voters, the attacker reaches a state where, in the result of the election, they are no longer compensated by the new votes. This allows the attacker to break privacy.

More precisely, the general idea of the proof is as follows. Consider an attacker $\mathcal{A}$ that breaks individual verifiability, *i.e.* wins the game $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{verif}}$ with non negligible probability. We construct an attacker $\mathcal{B}$ that breaks privacy, *i.e.* wins $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{priv},\beta}$. $\mathcal{B}$ starts by registering, and corrupting, the same voters as $\mathcal{A}$, using oracles $\mathcal{O}_{\mathsf{reg}}$ and $\mathcal{O}_{\mathsf{corr}}$. Let $id_1, \ldots, id_n$ be this first set of voters. $\mathcal{B}$ then registers another set of n voters $id'_1, \ldots, id'_n$, where the id'_i are fresh identities, that $\mathcal{A}$ does not use.

$\mathcal{B}$ then simulates $\mathcal{A}$, using the oracle $\mathcal{O}^p_{\mathsf{vote}}$ to simulate $\mathcal{A}$'s calls to $\mathcal{O}^v_{\mathsf{vote}}$. Specifically, when $\mathcal{A}$ calls $\mathcal{O}^v_{\mathsf{vote}}(id, v)$, $\mathcal{B}$ calls the oracle $\mathcal{O}^p_{\mathsf{vote}}(id, v, v^{\mathsf{blank}})$, where v^{blank} is a neutral vote. Once $\mathcal{B}$ is done simulating $\mathcal{A}$, it triggers the new voters id'_i to vote, by calling the oracle $\mathcal{O}^p_{\mathsf{vote}}(id'_i, v^{\mathsf{blank}}, v_i)$, where v_i is the (last) vote cast by id_i. The vote of each id'_i compensates the vote of id_i, so that the condition $\rho(\mathsf{V}_0) = \rho(\mathsf{V}_1)$ from $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{priv}}$ holds. $\mathcal{B}$ then obtains the result r of the election, which is equal to $r_1 * r_2$, where r_1 is the tally of the ballots cast by $\mathcal{A}$, and r_2 the tally of the additional ballots cast by $\mathcal{B}$. Then:

- if $\beta = 0$: then all the votes cast by the id'_i were v^{blank}, and the result is thus $r = r_1$. Since $\mathcal{A}$ breaks individual verifiability, r_1 does not contain the honest votes, *i.e.*, for all multiset V_c of votes, $r \neq \rho(v_1, \ldots, v_n) * \rho(V_c)$.

- if $\beta = 1$ however, the votes cast by the id'_i were the v_i, and the partial tally r_2 is therefore $r_2 = \rho(v_1, \ldots, v_n)$. Hence, the result r does contain the honest votes.

Therefore, by observing whether the final result of the election contains the honest votes, $\mathcal{B}$ is able to guess β, and wins $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{priv}}$.

We now give the detailed proof of the theorem.

Proof. Let $\mathcal{A} = \mathcal{A}_1, \mathcal{A}_2$ be an adversary that breaks individual verifiability, *i.e.* wins $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{verif}}$. We construct the adversary $\mathcal{B} = \mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ that attacks privacy, *i.e.* plays $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{priv},\beta}$, as follows.

- $\mathcal{B}_1^{\mathcal{O}_{\mathsf{reg}},\mathcal{O}_{\mathsf{corr}}}(\mathsf{pk})$ first simulates $\mathcal{A}_1^{\mathcal{O}_{\mathsf{reg}},\mathcal{O}_{\mathsf{corr}}}(\mathsf{pk})$. $\mathcal{B}_1$ registers and corrupts the same identities as $\mathcal{A}$ using its own oracles. $\mathcal{B}_1$ keeps a list I_1 of the identities it registers by calling $\mathcal{O}_{\mathsf{reg}}$. Once $\mathcal{A}_1$ returns, $\mathcal{B}_1$ calls $\mathcal{O}_{\mathsf{reg}}$ another $|I_1|$ times, on fresh identities that are simply bitstrings of length λ drawn uniformly and independently at random.It keeps a list I_2 of these fresh identities. Let us denote $\mathsf{U} = \mathsf{U}_1 \parallel \mathsf{U}_2$ the list of identities and credentials in the $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{priv},\beta}$ game at this point, where U_1 and U_2 respectively contain the credentials of the identities in I_1 and I_2. Note that the list D of dishonest identities is the same in $\mathsf{Exp}_{\mathcal{V},\mathcal{B}}^{\mathsf{priv},\beta}$ and the corresponding execution of $\mathsf{Exp}_{\mathcal{V},\mathcal{A}}^{\mathsf{verif}}$.

- $\mathcal{B}_2^{\mathcal{O}^p_{\mathsf{vote}},\mathcal{O}_{\mathsf{cast}}}(\mathsf{pk})$ maintains a list L, initially empty, which will be used to record the calls to $\mathcal{O}^p_{\mathsf{vote}}$. $\mathcal{B}_2$ internally runs $\mathcal{A}_2^{\mathcal{O}^v_{\mathsf{vote}},\mathcal{O}_{\mathsf{cast}}}(\mathsf{pk})$, answering its queries to the oracles as follows.
  - When $\mathcal{A}_2$ calls $\mathcal{O}^v_{\mathsf{vote}}(id, v)$, provided $id \in I_1$, $\mathcal{B}$ calls $\mathcal{O}^p_{\mathsf{vote}}(id, v, v_{\mathsf{neutral}})$. $\mathcal{B}_2$ then applies the revote policy to list $\mathtt{LL}$: it removes any previous couple (id, v') (for any v') from L, and then appends (id, v) to L. If $\mathcal{O}^p_{\mathsf{vote}}(id, v, v_{\mathsf{neutral}})$ returns a ballot b, $\mathcal{B}$ returns it to $\mathcal{A}_2$.
  - When $\mathcal{A}_2$ calls $\mathcal{O}_{\mathsf{cast}}(id, b)$, provided $id \in I_1$, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{cast}}(id, b)$.

If at any point during the simulation $\mathcal{A}_2$ blocks or fails, $\mathcal{B}$ stops the simulation. At this point, it is clear that if $\beta = 0$, $\mathcal{A}$ has been accurately simulated. Indeed in that case $\mathcal{A}$ is shown a board where the votes it wanted to cast have indeed been cast.

Let then Voted be the list of the votes in L: $\mathsf{Voted} = [v \text{ for } (id, v) \in L]$. Let also BB_a be the value of the board BB in game $\mathsf{Exp}^{\mathsf{priv},\beta}_{\mathcal{V},\mathcal{B}}$ at that point. By the previous observation, if $\beta = 0$, then BB_a is also equal to the value of BB in game $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}$ at this point.

Let us show that, except with a negligible probability that we will denote $p_1(\lambda)$, BB_a contains no ballots with identities from U_2, *i.e.* ballots b such that $\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U}) \in I_2$. There exists a polynomial $q(\lambda)$ that bounds $\mathcal{A}$'s running time. Consider a game $\mathsf{Exp}^{\mathsf{rand}}_{\mathcal{C}}(\lambda)$, where $q(\lambda)$ bitstrings of length λ are drawn uniformly independently at random, and the adversary $\mathcal{C}$ must propose a bitstring, and wins if this guess is one of the $q(\lambda)$ randomly drawn strings. It is clear that there no adversary $\mathcal{C}$ has a non-negligible probability to win that game. If $I_1 \cap I_2 \neq \emptyset$, an adversary $\mathcal{C}$ for game $\mathsf{Exp}^{\mathsf{rand}}$ could simply run $\mathcal{A}$ internally (running the Setup and $\mathsf{Register}$ algorithms by itself) and propose at random any of the (at most $q(\lambda)$) identities in I_1. Assimilating the identities drawn at random by $\mathcal{B}$ with those drawn by $\mathsf{Exp}^{\mathsf{rand}}$, $\mathcal{C}$ would win as soon as it picked an $id \in I_1 \cap I_2$. Thus, $I_1 \cap I_2 = \emptyset$ except with negligible probability.

Now, for each $b \in \mathsf{BB}_a$:

- either b was added to BB by a call $\mathcal{B}$ made to $\mathcal{O}^p_{\mathsf{vote}}$. By definition of $\mathcal{B}$, such a call is $\mathcal{O}^p_{\mathsf{vote}}(id, v_0, v_1)$ for some $id \in I_1$ and some votes v_0, v_1. Thus, $b = \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_\beta)$, and by assumption on $\mathsf{extract}_{id}$ we have $\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U}) = id \in I_1$. Thus, unless $I_1 \cap I_2 \neq \emptyset$, $id \notin I_2$.
- or b was added to BB by a call made by $\mathcal{B}$ to $\mathcal{O}_{\mathsf{cast}}(id, b)$ for some id. Hence, by definition of this oracle, $id \in \mathsf{D}$ and $\mathsf{Valid}(id, b, \mathsf{BB}, \mathsf{pk})$. Assume there exists such a b that does not satisfy $\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U}) \notin I_2$. Let us consider the first such ballot b_0 that is cast by $\mathcal{A}$. Consider an adversary $\mathcal{C}$ for $\mathsf{Exp}^{\mathsf{rand}}$, that runs an election by itself, *i.e.* running the Setup, $\mathsf{Register}$, *etc.* algorithms by itself. $\mathcal{C}$ does just as $\mathcal{B}$, except that it does not draw and register the identities in I_2: from the point of view of $\mathcal{C}$, the strings drawn by the game $\mathsf{Exp}^{\mathsf{rand}}$, that $\mathcal{C}$ does not have access to, will play the role of I_2. $\mathcal{C}$ thus only has access to U_1. $\mathcal{C}$ continues the execution of $\mathcal{B}$: the only point (before tallying) where $\mathcal{C}$ would need to use U_2 is when running the Valid algorithm to answer $\mathcal{B}$'s calls to $\mathcal{O}\mathsf{cast}$. $\mathcal{C}$ instead uses U_1 when that situation arises. $\mathcal{C}$ chooses at random one of the polynomially many calls to $\mathcal{O}_{\mathsf{cast}}(id, b)$ that $\mathcal{B}$ makes, and returns $\mathsf{extract}_{id}(id, b, \mathsf{sk}, \mathsf{U})$. With non-negligible probability, the call chosen was $\mathcal{O}_{\mathsf{cast}}(id_0, b_0)$. In that case, the execution of $\mathcal{C}$ accurately follows that of $\mathcal{B}$. Indeed, as explained, the only point where that execution might not be accurate is when $\mathcal{C}$ has run Valid using U_1 instead of $\mathsf{U}_1 \cup \mathsf{U}_2$. By assumption, up to that point, no ballots that extract to an identity in I_2 were submitted. By assumption, the validity of a ballot depends on no other credential that those of the id under which it is submitted and of the id it extracts to. Thus, only using U_1 up to the $\mathcal{O}_{\mathsf{cast}}(id_0, b_0)$ call yields the same execution as using $\mathsf{U}_1 \cup \mathsf{U}_2$. Therefore, the ballot returned by $\mathcal{C}$ is indeed the b_0 created by $\mathcal{A}$ such that $\mathsf{extract}_{id}(b_0, \mathsf{sk}, \mathsf{U}) \in I_2$, and $\mathcal{C}$ wins $\mathsf{Exp}^{\mathsf{rand}}$.

  This establishes that, except with negligible probability, all ballots b cast by $\mathcal{A}$ satisfy $\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U}) = id \notin I_2$.

$\mathcal{B}_2$ then calls $\mathcal{O}^p_{\mathsf{vote}}(id_1, v_{\mathsf{neutral}}, v_1)$, …, $\mathcal{O}^p_{\mathsf{vote}}(id_l, v_{\mathsf{neutral}}, v_l)$, where $v_1, \ldots, v_l$ are the elements of Voted, and $id_1, \ldots, id_l$ are pairwise distinct identities from I_2. Note that this is possible since all identities in L are distinct and in I_1: $l = |L|$ is hence smaller than $|I_1| = |I_2|$. Let then $\mathsf{BB}_b = [b_1, \ldots, b_l]$ be the set of new ballots added to the board by these l calls, *i.e.* ballots for v_{neutral} if $\beta = 0$ and $v_1, \ldots, v_l$ if $\beta = 1$.

At this point, we have $\mathsf{BB} = \mathsf{BB}_a \parallel \mathsf{BB}_b$. By construction, except with negligible probability $p_2(\lambda)$, BB_b only contains ballots with identities from U_2, *i.e.* ballots b such that $\mathsf{extract}_{id}(b, \mathsf{sk}, \mathsf{U}) \in I_2$.

$\mathcal{B}_2$ then returns.

- $\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{V}}$ will then check that $\rho(\mathsf{V}_0) = \rho(\mathsf{V}_1)$, where V_0 and V_1 are the lists it keeps, which contain the last votes $\mathcal{O}^p_{\mathsf{vote}}$ has been called on for each id. Considering the definition of $\mathcal{B}_2$, at this point we always have

$$\rho(\mathsf{V}_0) = \rho(\mathsf{V}_1) = \rho(v_1, \ldots, v_l, \underbrace{v_{\mathsf{neutral}}, \ldots, v_{\mathsf{neutral}}}_{l \text{ times}})$$

  Hence this check necessarily succeeds, and $\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{V}}$ computes $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U})$.

- $\mathcal{B}_3$ obtains a result r. If $r = \perp$, $\mathcal{B}_3$ returns 1. If $\mathcal{A}_2$ blocked previously during its simulation, $\mathcal{B}_3$ also returns $\beta' = 1$. Indeed, as explained earlier, when $\beta = 0$, $\mathcal{A}$ is accurately simulated. Hence, intuitively, whenever $\mathcal{A}$ wins $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V}}$, the simulated $\mathcal{A}_2$ failing means that $\beta = 1$.

  Otherwise, $\mathcal{B}$ computes $\mathsf{D}(r, \mathsf{Voted})$ and:

  - if $\exists V_c.\ r = \rho(\mathsf{Voted}) * \rho(V_c)$ then $\mathcal{B}$ returns $\beta' = 1$
  - otherwise, $\mathcal{B}$ returns $\beta' = 0$.

We will now prove that if $\mathcal{A}$ breaks individual verifiability, then $\mathcal{B}$ wins the privacy game except with negligible probability.

Let us first establish that if $\mathsf{ValidTally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1)$ holds, then $\mathsf{ValidTally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U}_1 \cup \mathsf{U}_2)$ also holds, except with negligible probability.

We construct an adversary $\mathcal{C}$, who plays the strong correctness game $\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V}}$ as follows.

- $\mathcal{C}_1$ is identical to $\mathcal{B}_1$.

- $\mathcal{C}_2$ first draws at random a bit β'', and then simulates $\mathcal{B}_2$ up to the point where $\mathcal{B}_2$ has finished simulating $\mathcal{A}_2$. $\mathcal{C}_2$ keeps a list BB, initially empty. It simulates each call to $\mathcal{O}^p_{\mathsf{vote}}(id, v_0, v_1)$ $\mathcal{B}$ makes by calling $\mathcal{O}^{vt}_{\mathsf{vote}}(id, v_{\beta''})$, and appending the obtained ballot to BB. It simulates each call to $\mathcal{O}_{\mathsf{cast}}(id, b)$ by appending b to BB, provided id is corrupted and $\mathsf{Valid}(id, b, \mathsf{BB}, \mathsf{pk})$.

- Once $\mathcal{C}_2$ has finished simulating $\mathcal{B}_2$, it draws at random a number $k \in [\![1, l]\!]$ Recall that l is the number of different ids $\mathcal{O}^p_{\mathsf{vote}}$ (and thus $\mathcal{O}^{vt}_{\mathsf{vote}}$) has been called on, and is also the number of additional calls to $\mathcal{O}^p_{\mathsf{vote}}$ $\mathcal{B}$ has yet to make. Note that, at this point, BB in $\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V},\mathcal{C}}$ is the same as BB_a in $\mathsf{Exp}^{\mathsf{priv},\beta''}_{\mathcal{V},\mathcal{B}}$. $\mathcal{C}$ then simulates the next $k-1$ calls to $\mathcal{O}^p_{\mathsf{vote}}(id, v_0, v_1)$, again by calling $\mathcal{O}^{vt}_{\mathsf{vote}}(id, v_{\beta''})$. If the kth call is $\mathcal{O}^p_{\mathsf{vote}}(id, v_0, v_1)$, $\mathcal{C}$ returns $(\mathsf{BB}, id, v_{\beta''})$.

The adversary $\mathcal{C}$ is polynomial, *i.e.* there exists a polynomial $q(\lambda)$ bounding its number of operations.

For any β, assume $\mathsf{ValidTally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1)$ holds and $\mathsf{ValidTally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U}_1 \cup \mathsf{U}_2)$ does not. Let us write $\mathsf{U}_2 = [(id_1, c_1), \ldots, (id_l, c_l)]$. Hence, there exists a smallest $k \in [\![1, l]\!]$ such that $\mathsf{ValidTally}(\mathsf{BB}_a \uplus [b_1, \ldots, b_{k-1}], \mathsf{sk}, \mathsf{U}_1 \cup \mathsf{U}'_2)$ holds and $\mathsf{ValidTally}(\mathsf{BB}_a \uplus [b_1, \ldots, b_k], \mathsf{sk}, \mathsf{U}_1 \cup \mathsf{U}'_2 \cup [(id_k, c_{id_k})])$ does not, where $\mathsf{U}'_2 = [(id_1, c_1), \ldots, (id_{k-1}, c_{k-1})]$. by definition, b_k has been added to BB_b by the kth call to $\mathcal{O}^p_{\mathsf{vote}}$ made by $\mathcal{B}$ after it had finished running $\mathcal{A}$. Therefore, $b_k = \mathsf{Vote}(\mathsf{pk}, id_k, c_k, v_\beta)$ for some v_β. Thus, provided $\mathcal{C}$ correctly guesses $\beta'' = \beta$ and k, $\mathcal{C}$ returns $(\mathsf{BB}_a \uplus [b_1, \ldots, b_{k-1}], id_k, v_\beta)$. The conditions on BB in $\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V},\mathcal{C}}$ holds, and thus $\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V},\mathcal{C}} = 1$. Therefore, $\mathsf{ValidTally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1)$ holds and $\mathsf{ValidTally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U}_1 \cup \mathsf{U}_2)$ does not with probability at most $2l\,\mathbb{P}\left[\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V},\mathcal{C}} = 1\right]$, which is smaller than $2q(\lambda)\,\mathbb{P}\left[\mathsf{Exp}^{\mathsf{ValidTally}}_{\mathcal{V},\mathcal{C}} = 1\right]$ since $l \leq q(\lambda)$. Let us denote $p_3(\lambda)$ this probability. By the strong correctness assumption, it is negligible.

As noted earlier, BB_a contains no ballots cast for identities in I_2 (except with probability $p_1(\lambda)$) and BB_b only ballots for identities in I_2 (except with probability $p_2(\lambda)$). In addition, $I_1 \cap I_2 = \emptyset$, except with some negligible probability $p_4(\lambda)$. Thus, by the piecewise tallying assumption, if $\mathsf{ValidTally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U}_1 \cup \mathsf{U}_2)$, then we have (regardless of β)

$$r = \mathsf{Tally}(\mathsf{BB}, \mathsf{sk}, \mathsf{U}) = \mathsf{Tally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U}) = \mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}) * \mathsf{Tally}(\mathsf{BB}_b, \mathsf{sk}, \mathsf{U})$$

In addition, by assumption, registering more users does not change the tally, *i.e.* $\mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}) = \mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1)$.

Hence, if $\mathsf{ValidTally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U}_1 \cup \mathsf{U}_2)$, then

$$r = \mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1) * \mathsf{Tally}(\mathsf{BB}_b, \mathsf{sk}, \mathsf{U})$$

except with some negligible probability $p_5(\lambda)$.

- If $\beta = 0$: then $\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{V},\mathcal{B}}(\lambda) = 1$ if and only if $\mathcal{B}_3$ returns 1 in this game, which happens either when $\mathcal{A}$ (simulated by $\mathcal{B}$) blocks, or when it does not and $\exists V_c.\ r = \rho(\mathsf{Voted}) * \rho(V_c)$ or $r = \bot$.

  Assume $\mathsf{ValidTally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1) \Rightarrow \mathsf{ValidTally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U})$, which, as we have established, holds except with probability $p_3(\lambda)$.

  Let us first examine the case where $\mathcal{A}$ does not block and $r \neq \bot$. Since $r \neq \bot$, $\mathsf{ValidTally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1)$ holds. Hence, $\mathsf{ValidTally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U})$ also holds, and as explained previously we thus know that $r = \mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1) * \mathsf{Tally}(\mathsf{BB}_b, \mathsf{sk}, \mathsf{U})$ (except with probability $p_5(\lambda)$). Since $\beta = 0$, BB_b only contains ballots for v_{neutral}. Hence, by consistency of the voting scheme, $\mathsf{Tally}(\mathsf{BB}_b, \mathsf{sk}, \mathsf{U}) = \rho(v_{\mathsf{neutral}})^l = \rho(v_{\mathsf{neutral}})$. Thus $r = \mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1)$.

  The condition $\exists V_c.\ r = \rho(\mathsf{Voted}) * \rho(V_c)$ is therefore equivalent to $\exists V_c.\ \mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1) = \rho(\mathsf{Voted}) * \rho(V_c)$. Since, in this case, $\mathcal{A}$ has been accurately simulated without blocking, does not return $\bot$, and BB_a is the board after its execution, this is exactly the condition under which $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda)$ does not return 1.

  Hence $\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{V},\mathcal{B}}(\lambda) = 1$ if and only if either $\mathcal{A}$ (simulated by $\mathcal{B}$) blocks, or constructs a board whose tally is $\bot$, or it does not and $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda) \neq 1$.

Since $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda)$ also does not return 1 when $\mathcal{A}$ blocks or when the tally is $\perp$, this implies that (except with probability at most $p_3(\lambda)+p_5(\lambda)$) $\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{V},\mathcal{B}}(\lambda) = 1$ if and only if $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda) \neq 1$. Thus

$$\left|\mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{V},\mathcal{B}}(\lambda) = 1\right] - \mathbb{P}\left[\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda) \neq 1\right]\right| \leq p_3(\lambda) + p_5(\lambda). \tag{5.2}$$

- If $\beta = 1$: then $\mathsf{Exp}^{\mathsf{priv},1}_{\mathcal{V},\mathcal{B}}(\lambda) = 1$ if and only if $\mathcal{B}_3$ returns 1 in this game, which happens either when $\mathcal{A}$ (simulated by $\mathcal{B}$) blocks, or when it does not and $\exists V_c.\ r = \rho(\mathsf{Voted}) * \rho(V_c)$ or $r = \perp$.

  Assume $\mathsf{ValidTally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1) \Rightarrow \mathsf{ValidTally}(\mathsf{BB}_a \uplus \mathsf{BB}_b, \mathsf{sk}, \mathsf{U})$, which, as we have established, holds except with probability $p_3(\lambda)$. Let us first examine the case where $\mathcal{A}$ does not block and $r \neq \perp$. As in the $\beta = 0$ case, we thus have $r = \mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1) * \mathsf{Tally}(\mathsf{BB}_b, \mathsf{sk}, \mathsf{U})$ (except with probability $p_5(\lambda)$). Since $\beta = 1$, BB_b contains ballots for $v_1, \ldots, v_l$, *i.e.* for Voted. Hence, by consistency, $\mathsf{Tally}(\mathsf{BB}_b, \mathsf{sk}, \mathsf{U}) = \rho(\mathsf{Voted})$, and therefore $r = \mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1) * \rho(\mathsf{Voted})$. By definition of Tally, there exists V such that $\mathsf{Tally}(\mathsf{BB}_a, \mathsf{sk}, \mathsf{U}_1) = \rho(V)$. Hence the condition $\exists V_c.\ r = \rho(\mathsf{Voted}) * \rho(V_c)$ necessarily holds.

  Therefore, except with probability at most $p_3(\lambda) + p_5(\lambda)$, $\mathsf{Exp}^{\mathsf{priv},1}_{\mathcal{V},\mathcal{B}}(\lambda) = 1$ if and only if either $\mathcal{A}$ (simulated by $\mathcal{B}$) blocks, or it does not and returns a board whose tally is $\perp$, or it does not and returns a board whose tally is not $\perp$. Hence

$$1 - \mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},1}_{\mathcal{V},\mathcal{B}}(\lambda) = 1\right] \leq p_3(\lambda) + p_5(\lambda). \tag{5.3}$$

We thus have, using 5.2 and 5.3:

$$\begin{aligned}
\mathbb{P}\left[\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda) = 1\right] &= \left(\mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{V},\mathcal{B}}(\lambda) = 1\right] - \mathbb{P}\left[\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda) \neq 1\right]\right) \\
&\quad + \left(\mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},1}_{\mathcal{V},\mathcal{B}}(\lambda) = 1\right] - \mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{V},\mathcal{B}}(\lambda) = 1\right]\right) \\
&\quad + \left(1 - \mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},1}_{\mathcal{V},\mathcal{B}}(\lambda) = 1\right]\right) \\
&\leq \left|\mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},1}_{\mathcal{V},\mathcal{B}}(\lambda) = 1\right] - \mathbb{P}\left[\mathsf{Exp}^{\mathsf{priv},0}_{\mathcal{V},\mathcal{B}}(\lambda) = 1\right]\right| + 2(p_3(\lambda) + p_5(\lambda))
\end{aligned}$$

Therefore, if $\mathcal{A}$ breaks individual verifiability, *i.e.* if $\mathbb{P}\left[\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V},\mathcal{A}}(\lambda) = 1\right]$ is not negligible, then $\mathcal{B}$ breaks privacy, which proves the theorem. □

Note that, in these reductions, for each *id* in U_1, $\mathcal{B}$ makes at most as many calls to $\mathcal{O}^p_{\mathsf{vote}}$ as $\mathcal{A}$ makes to $\mathcal{O}^v_{\mathsf{vote}}$, and at most as many calls to $\mathcal{O}_{\mathsf{cast}}$ as $\mathcal{A}$ makes to $\mathcal{O}_{\mathsf{cast}}$. In addition, for each *id* in U_2, $\mathcal{B}$ makes at most one call to $\mathcal{O}^p_{\mathsf{vote}}$, and no call to $\mathcal{O}_{\mathsf{cast}}$. Thus, the exact same proof proves that the result also holds if both the games $\mathsf{Exp}^{\mathsf{priv}}_{\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{verif}}_{\mathcal{V}}$ are modified to prevent revote, by allowing only one call to $\mathcal{O}^p_{\mathsf{vote}}/\mathcal{O}^v_{\mathsf{vote}}$ and/or to $\mathcal{O}_{\mathsf{cast}}/\mathcal{O}_{\mathsf{cast}}$ for each *id*.

We have shown that, with a standard game-based definition for privacy and an intuitive notion of individual verifiability, under some mild assumptions on the voting scheme, privacy implies individual verifiability.

As we already observed in the previous chapter regarding symbolic models, this implication only holds when considering the same trust assumptions *w.r.t.* the election authorities for both properties. In fact, Benaloh's definition for privacy assumes a honest ballot box. In the game, as soon as a ballot is emitted by a honest voter, it is added to the ballot box, and cannot be

removed by the adversary: the adversary can only add new ballots to the ballot box, but not control it. For this reason, we considered a definition for verifiability that also assumes a honest ballot box. This assumption is actually shared by almost all existing game-based definitions for vote privacy. As we discuss in the following Chapter, this highlights a gap in existing definitions. Indeed, voting protocols typically aim at being private without having to trust the ballot box, but the privacy notions usually considered assume it to be honest, and are thus too weak.

Conclusion on Privacy and Verifiability

In Chapters 4 and 5, we have shown a subtle relation between privacy and verifiability, namely that privacy implies individual verifiability, which is rather counter-intuitive. Our result holds in a cryptographic as well as a symbolic setting, for various trust assumptions.

As we have explained, if an adversary who controls the ballot box wishes to learn the vote of some voter Alice, he can simply remove all ballots except Alice's from the list of recorded ballots, and replace all of them with encryptions of valid votes of his choice. When seeing the result of the election, the adversary, who already knows the value of all votes except Alice's – he chose them himself – will deduce how Alice voted.

As noted in previous chapters, our result holds when the same trust assumptions are made regarding the election authorities, *i.e.* when the same entities (ballot box, channels, tallying authority, *etc.*) are trusted for privacy and verifiability. Our findings thus point out that if voters wish for their vote to be private without having to trust some authorities, then the system must also ensure that their votes are counted without trusting these same authorities. That is, an entity that has the power to cheat on the result, and make it so that the votes are not counted correctly, also has the power to learn information about how voters voted, even for entities that do not have access to the secret keys.

Limitations. Individual verifiability is only one part of verifiability. It does not account for universal nor eligibility verifiability. So our result cannot be used to conclude that a private voting scheme ensures all desirable verifiability properties. Instead, it demonstrates that there is no hope to design a private voting system if it does not include some degree of verifiability, namely individual verifiability at least.

Our result assumes counting functions that have the partial tally property. The proof technique does not extend immediately to more complex counting functions such as STV or Condorcet, but as we will see in the next chapter, the implication still holds at least in some cases without this assumption. Also, our results implicitly discard anonymous channels: our computational model does not account for anonymous channels while our election determinism assumption discards at least some use of anonymous channels. Intuitively, in presence of such channels, an attacker may be able to modify a ballot without being able to tell which one, hence breaking verifiability without breaking privacy. It would be interesting to identify which kind of anonymous channels and, more generally, which form of non determinism can still be tolerated.

Dishonest ballot box. As discussed earlier (see introduction of Part II), the apparent contradiction between our result and the fact that known protocols have been proved private without

being verifiable (*e.g.* Helios without verification steps) is explained by closely looking at the trust assumptions.

Our main theorem states that privacy implies individual verifiability *with the same trust assumptions*. Verifiability is typically studied against a dishonest ballot box. However, the privacy definition introduced by Benaloh assumes an honest ballot box, as well as most existing game-based privacy definitions of the literature [26] that are used to study protocols. Therefore, our main theorem in the computational model shows that whenever a voting scheme is private *w.r.t.* an honest ballot box, then it is also individually verifiable *w.r.t. an honest ballot box*, which is of course a rather weak property.

This highlights a crucial need for a ballot privacy definition in the context of a dishonest ballot box, in a cryptographic setting. So far, privacy has only been proved assuming an honest ballot box, which forms a very strong trust assumption that was probably never made clear to voters nor election authorities: voting schemes aim at being private *without needing to trust the ballot box*.

As already pointed out in Introduction of Part II, defining privacy against a dishonest ballot box is hard, as naïvely adapting usual definitions typically yields properties that are too strong, and can never be satisfied. This problem is the subject of the next chapter.

Chapter 6

Privacy against a Dishonest Ballot Box

6.1 Introduction

Electronic voting systems usually try to ensure that the votes remain private. Typically, votes are encrypted with a public key, and the associated decryption key is shared among several trustees, so that at least a certain number of them are required in order to decrypt and compute the tally. In particular, a typical design choice is not to give access to this secret key to the ballot box, *i.e.* the server in charge of collecting the ballots from all voters. The idea behind this choice is that it should ensure that even if this server is not trusted, it should not be able to learn any information on the votes, *i.e.* to break the voters' privacy.

However, as we pointed out in the previous chapters, this trust assumption is not captured properly by existing game-based security definitions. A similar observation has been made *e.g.* in [67, 29]. In brief, existing definitions (*e.g.* [25, 26, 50]) consider a game where the adversary controls the votes cast by honest parties but cannot control the resulting ballots: these get placed on the bulletin board before it is tallied and cannot ever be modified or removed. In other words, current security notions allow to prove security of schemes only under the assumption that the bulletin board is honest, even if they are designed (and claimed) to resist a dishonest voting server.

This gap between security goals and security definitions has recently been confirmed by Roenne [94] in the case of Helios [10]. As already described *e.g.* in Chapter 3, the attack shows that a malicious board can break privacy of votes if users are allowed to revote. Furthermore, it seems non-trivial to prevent it even if voters and external auditors carry out additional checks (*e.g.* forbidding duplicate ballots, a.k.a. weeding). Even detecting the attack would require unrealistic countermeasures where every voter carefully records any of her ballots, even the ones that failed to reach the ballot box.

In this chapter we propose a way to fill that gap. We design novel security definitions which allow the rigorous study of privacy of electronic voting in presence of a malicious bulletin board, study relations between the notions which we propose, and apply them to several well-established schemes.

In [64], we made a first attempt at modelling privacy against a dishonest board, where we assumed that all honest voters perform the verifications specified by the protocol. Such an assumption was unrealistic: in a realistic setting, it is more likely that a (small) fraction of honest voters perform the required tests while the others stop after casting their vote. The notion of

privacy we propose in this chapter is much more precise and general than that initial definition, which we will thus not present here.

Before we outline our results it is useful to discuss the security of Helios against an adversary who can control the bulletin board. Recall that in Helios voters encrypt their vote and send their encrypted ballot to a bulletin board, that displays them. The tally is done through mixnets or using the homomorphic property of the encryption. Importantly, voters can and should in fact check that their ballot appears on the bulletin board. First, notice that by changing the board an adversary may influence the final result and indirectly learn information about the votes of the honest party. This can be easily seen by considering the extreme case, already discussed earlier, where the adversary culls all but one ballot from the board: the tally then reveals the value of the underlying vote. The following three scenarios show that different usages of Helios, defined by how voters cast/check their ballots, lead to different security guarantees.

- The scheme offers the strongest guarantees when all honest voters vote and check that their vote appears on the ballot box. In this case the result of the election contains all of the honest votes plus at most as many votes as the number of voters controlled by the adversary. However, such assumptions on honest voters are unrealistic – in practice, only a small fraction of voters perform the checks.

- Assume now that not all honest voters vote. Then the security of Helios decreases. Indeed, a malicious board may use absentee voters to place ballots of her choice. So the privacy of the election is as good as what an attacker can learn from a result formed from the honest voters that did vote and any choice of votes from the remaining voters (dishonest or not).

- Finally, the most common scenario is that not all honest voters vote and only a fraction of them actually conduct the suggested verification steps. In this case the security of Helios decreases further. Indeed, a malicious board may now selectively remove ballots that have been cast by voters that do not check. Hence a malicious board has now even more control on the result, that may leak more information since it may contain fewer votes (at the will of the adversary).

What does this example tell us? The strongest security is only achieved under unreasonable assumptions and more plausible uses lead to weaker guarantees. Furthermore, the level of control the adversary has on the board (and therefore over the verifiability of the scheme/the election result) may vary considerably not only between schemes, but even between different usage scenarios of *the same* scheme. Therefore, as already hinted at in the previous chapters, taking into account the verification mechanisms, and the fact that only some of the voters would actually perform them, seems inevitable if one aims to characterise the privacy of a voting scheme. One crucial conclusion of this brief discussion is that it is difficult to pin down "the right" level of privacy. Instead, an appropriate security definition should account for the possibility of a spectrum of privacy levels derived from varied trust assumption and usage scenarios.

Another important observation is that the distinction between the different levels of guarantees can be quite subtle, and the associated security implications difficult to evaluate. From this perspective, not only it is difficult to define security but it is also very hard to compare voting schemes and select the ones that should be used, if a malicious board is a concern. All of these concerns highlight the need for a framework that allows for a systematic analysis of existing schemes.

Contributions and outline of the chapter. We make several contributions which address the challenges outlined above. We first provide *a family* of rigorous game-based definitions for ballot privacy against malicious bulletin boards. Different instantiations capture different levels of ballot privacy and in particular we can accurately capture the distinct levels of security discussed above.

Next, we introduce a family of ideal functionalities for defining simulation-based security. Over the typical functionality which collects the votes and applies a result function, our notions permit adversarial control over the result of the computation (see below for the details). These functionalities specify precisely the expected abilities (and therefore the limitations) of an adversary who tampers with the ballot box, so that the resulting level of security can be more easily evaluated than for game based security definitions.

We then relate these two families of privacy notions: we show that under mild assumptions the former imply the latter. Our proof for this implication is generic in that we do not relate members of the two families of definitions instance by instance.

Then, we use our definitions to study the security of several well established protocols.

One aspect of our work which is worth discussing is that it reflects the subtle interplay between verifiability and privacy exposed in the previous chapters. Our different flavours of ideal functionalities reflect different adversarial capabilities to modify honest votes. Is this a privacy or a verifiability property? Intuitively, ballot privacy says that the adversary should not learn more information about the votes than the result itself. Hence, whether or not the adversary can remove or alter honest votes, or add more votes, gives more control to the adversary over the result, which interferes with the level of privacy offered by the voting scheme. Actually, we show that our privacy notion, as soon as a sufficiently strong instance of it is considered, implies individual verifiability against a dishonest ballot box. This confirms the results from the previous chapters. Interestingly, this proof is completely different than the previous ones: we leverage the relation we prove between our notion and simulation-based security to prove directly in the ideal world that individual verifiability holds, which is much easier.

Finally, we try to get a better insight on how to compare different instances of our family of ideal functionalities, and on what level of privacy they guarantee. To do this we quantify the amount of information they allow to be leaked to the adversary, by exploring the notion of entropy. Following the formalism proposed in [27], we consider votes as random variables, and assume the adversary is trying to make some specific observation on them. This observation is expressed by a *target function*, *i.e.* a function that associates a list of votes to some value intended to represent the information the adversary tries to determine. We then compute the conditional entropy on the target function (applied to the votes), given the view of the adversary, *i.e.* all the messages it receives and his internal state. We compare that entropy for different ideal functionalities, and discuss our findings.

More details about our privacy notion. We build upon the security notion BPRIV introduced in [26]. To understand our approach it helps to recall the general idea behind BPRIV, and more generally existing game-based definitions. Essentially, the adversary has to distinguish between two situations. The first is a situation where honest voters submit ballots containing votes selected by the adversary. The second is where the ballots submitted by the honest voters are "fake". In BPRIV these are simply ballots of some value, also chosen by the attacker. In the process the adversary is allowed to learn the result of the tally process. Ignore for the moment the ballots cast by dishonest parties. In the case of an honest bulletin board (where all ballots are placed on the bulletin board to be tallied) we always return to the adversary the

tally corresponding to the real ballots. If no adversary can tell the two situations apart then the ballots themselves, together with the result learned by the adversary, do not leak information about the underlying votes.

We capture security against an adversary who can tamper with the bulletin board, by requiring that there exists a *recovery* algorithm which, essentially, can detect *how* the adversary has tampered with the ballots issued by the honest users. The output of the recovery algorithm can be thought a small program, written in a small programming language with commands that act on the bulletin board (delete honest votes, modify votes, re-order votes, *etc.*). If such a recovery algorithm exists, we return to the adversary the tally of the real ballots but after the tampering program has been applied. A voting scheme is said to satisfy mb-BPRIV – shorthand for "BPRIV against a malicious ballot box" – when no adversary can distinguish the real world from the fake one.

The typical definition for simulation based security involves a functionality which collects the list of votes of all parties (honest and corrupt) and simply returns the result of the election determined by the list. To capture the setting where an adversary can to some extent tamper with the bulletin board (and therefore the list of votes that is tallied) we modify the ideal functionality to reflect this adversarial ability. We now give a high level (and imprecise) sketch of how we proceed as follows. The functionality is parametrised by a what we could call a "guard" P, that is, a predicate that limits what power the functionality gives to the adversary. After it collects the votes, but before it returns the result, the functionality allows the adversary to tamper with the list of votes via an arbitrary program f. However, it will only accept to compute the result if that program satisfies the predicate P. Roughly, showing that a voting scheme implements this ideal functionality for a given guard P therefore guarantees that no adversary against the scheme can have more power than what P specifies.

We can then establish a link between security *w.r.t.* mb-BPRIV parametrised by recovery algorithm f and idealised security with respect to an ideal functionality parametrised by guard P. If f and P are compatible, that is, roughly, if the tampering that f can detect are allowed by P, then any scheme which is mb-BPRIV secure and strongly consistent – a notion introduced in [26] – is secure with respect to the ideal functionality. As observed by [26], mb-BPRIV alone is not sufficient for simulation-based security, as it does not account for the case where a badly conceived tallying algorithm leaks some unwanted information on the votes: indeed, since the tally is always computed on the real ballots, such a leakage would not help the adversary to distinguish between the real and fake ballots, *i.e.* to break mb-BPRIV. Hence [26] introduced the notion of strong consistency, which demands that the tally reveals only the desired result function on the votes (and no additional information).

Related work. The first game-based definition which considers a malicious board has been proposed by Bernhard and Smyth [29]. Their definition extends Benaloh's approach [25]. The adversary submits a board and the tally is performed only if the ballots on the board that come from honest voters are such that the subtally does not differ in the "left" and "right" worlds. This somehow corresponds to one possible instance of our recovery algorithm in mb-BPRIV, where the attacker may remove any honest vote and add an arbitrary number of votes, independently of whether honest voters do check their ballot and independently of the number of dishonest voters. Note that [29] requires that ballots cannot be modified at all (*e.g.* they cannot contain a tag such as the date). Perhaps the most important technical difference from our approach is that Benaloh's definitional approach does not seem to allow a formal link with a simulation-based notion of security. In brief, Benaloh's definition does not allow to construct a simulator which

can simulate on the fly a fake board towards an adversary: the global consistency requirement between the subtally of the real votes seems to preclude an on-line simulator which can fake a board. Furthermore, this approach assumes that the counting function admits partial tally, which discards many modern counting functions such as Condorcet or STV.
The shortcomings that stem from the use of Benaloh's approach are also shared by [64]. This definition can be again seen as an extension of Benaloh's definition but which assumes that *all* honest voters check that their ballot appears on the bulletin board.
Recently, Bursuc, Dragan and Kremer [40] have studied the security of encryption schemes where ballots can be partially modified, for example by a malicious device. They propose a variant of BPRIV that accounts for such behaviours. The case of a malicious board corresponds to the case where ballots can be fully modified. Then for malicious boards, vote privacy defined in [40] can be seen as an instance of mb-BPRIV where the recovery algorithm lets the adversary tamper arbitrarily with the honest ballots. However, in such a case, all schemes would be declared insecure. So the model of [40] does not seem suitable to reason about malicious boards in general. Instead, it addresses a class of schemes where security is due to the part of the ballot that is securely transmitted to the (honest) board, despite the adversary tampering with the other parts of the ballot.

6.2 Background

The computational model we consider for voting protocols is the same as in the previous chapter, with a few extensions and additional assumptions that we describe in this section. We also present the original BPRIV notion that we build upon.

6.2.1 Notations

In the previous chapter, corruption of voters was dynamic: the adversary had access to a registration oracle to generate credentials for any number of voters, and a corruption oracle, to corrupt any of them.

Instead, we now fix beforehand the identities of voters. We consider a finite set $\mathcal{I} = \mathsf{H} \cup \mathsf{D}$ of voter identities, partitioned into the sets H and D of honest and dishonest voters. H is further partitioned into sets $\mathsf{H}_{\mathsf{check}}$ and $\mathsf{H}_{\overline{\mathsf{check}}}$, meant to contain the identities of voters who verify their vote (resp. do not verify).

The interest of fixing voter identities will become clear later in the chapter. Basically, we will relate two notions of vote privacy, showing that one implies the other. Having fixed the sets of honest and corrupted voters beforehand will then let us know that this implication holds *when considering the same number of adversary-controlled voters* for both notions, which would not be the case if the voters were dynamically corrupted. In a model with dynamic corruption, we could only prove the weaker claim that the implication holds with potentially different numbers of corrupted voters for each notion.

We study schemes for which the adversary can tamper with the bulletin board, which, as before, is simply a list of ballots. We make an additional mild assumption regarding the format of the ballots: we assume them to be pairs of bitstrings.

Definition 44 (Bulletin board)**.** *A* bulletin board BB *is a list of pairs of bitstrings which we call* ballots*. For a ballot* (p, b)*, we call the bitstring* p *a* public credential *and the bitstring* b *is a ciphertext.* $\mathsf{BB}[j]$ *denotes the* j*th element of* BB*.*

We will also call extended bulletin board *a board where elements are associated to an identity,* i.e. *a list of elements of the form* $(id, (p, b))$.

While the exact nature of the ballots, *i.e.* public credentials and ciphertexts, depends on the protocol considered, we intuitively intend the public credential to link to the identity of the voter who casts the ballot, and the ciphertext to contain the vote expressed by the ballot.

As in the previous chapters, we call *counting functions* the functions which calculate the result of an election. To be more general, we no longer assume a partial tallying property. In addition, we will no longer explicitly encode the revote policy in the games defining properties (*e.g.* having ballots cast by oracles replace previous ballots, to encode that the last vote counts). Instead, the revote policy will be encoded in the counting function. That is, we now consider counting functions that 1. take the identities of voters into account, 2. apply to lists rather than multisets of votes. This way, any revote policy can be expressed by a counting function. We will express our results generically (for an arbitrary unspecified counting function ρ), so that they are independent of the choice of a particular revote policy.

Definition 45 (Counting function)**.** *A* counting function ρ *is a mapping that takes as input a sequence* S *of pairs* (id, v)*, where* $id \in \mathcal{I}$ *and* v *is a vote, and returns a value (bitstring) intended to represent the result of tallying the votes in* S*. It may use the ids contained in* S *to apply a revote policy.*

We assume a special value $\perp$*, intended to represent an invalid identity and vote (*e.g. *obtained by decrypting a ballot that was incorrectly generated), that should not be counted. Formally, we require that counting functions ignore this value,* i.e. *that for all* l*,* l'*,* $\rho(l||\perp||l') = \rho(l||l')$.

6.2.2 Original BPRIV notion

As mentioned in introduction, the privacy notion we propose in the next section builds upon the definition for ballot privacy BPRIV introduced by Bernhard et al [26]. To provide some context, in this section we briefly present this existing definition (adapted to the notations and model we use here), as well as some associated properties. Finally we recall the result from [26], which states that BPRIV together with the associated notions imply a simulation-based notion of privacy. We take this opportunity to make explicit an assumption required for this result that was missing from the original paper.

Ballot privacy states that ballots themselves do not reveal information about the underlying votes (even after tallying).

The original BPRIV notion models an honest ballot box whereas we consider a malicious one. To distinguish between the two notions we will from now on refer to the original one as hb-BPRIV security (for "honest ballot box") and to the notion we introduce in the next section as mb-BPRIV (for "malicious ballot box").

hb-BPRIV. Essentially, in hb-BPRIV, the adversary is interacting with an execution of the voting protocol, and sees either the real ballots, or fake ballots containing fake votes, depending on a secret bit β. Using oracles, he can choose the values of the real and fake votes, and cast any ballot he can construct (in the name of a corrupted voter). In the end, regardless of β, the adversary is shown the result of tallying the real ballots. He must then guess β, and the privacy property is that he should not be able to correctly do so. Intuitively, this notion of ballot privacy

$\mathsf{Exp}^{\mathsf{hb\text{-}BPRIV},\beta}_{\mathcal{A},\mathcal{V}}(\lambda)$

$\mathsf{BB}_0, \mathsf{BB}_1 \leftarrow \emptyset$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $c \leftarrow \mathsf{Register}(1^\lambda, id)$
  $\mathsf{U}[id] \leftarrow c, \mathsf{PU}[id] \leftarrow \mathsf{Pub}(c)$
for all $id \in \mathsf{D}$ **do** $\mathsf{CU}[id] \leftarrow \mathsf{U}[id]$
$\mathcal{A}^{\mathcal{O}\mathsf{voteLR}}(\mathsf{pk}, \mathsf{CU}, \mathsf{PU})$
$d \leftarrow \mathcal{A}^{\mathcal{O}\mathsf{tally}_{\mathsf{BB}_0,\mathsf{BB}_1}}()$
output d.

$\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ for $id \in \mathsf{H}$

$(p_0, b_0) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$
$(p_1, b_1) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_1)$
if $\mathsf{Valid}(id, (p_\beta, b_\beta), \mathsf{BB}_\beta, \mathsf{pk}) = \bot$
then return $\bot$
else $\mathsf{BB}_0 \leftarrow \mathsf{BB}_0 \parallel (p_0, b_0); \mathsf{BB}_1 \leftarrow \mathsf{BB}_1 \parallel (p_1, b_1)$
return (p_β, b_β).

$\mathcal{O}\mathsf{tally}_{\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 0$

$(r, \Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}_0, \mathsf{sk})$
return (r, Π)

$\mathcal{O}\mathsf{tally}_{\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 1$

$(r, \Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}_0, \mathsf{sk})$
$\Pi' \leftarrow \mathsf{SimProof}(\mathsf{BB}_1, r)$
return (r, Π')

Figure 6.1 – The hb-BPRIV game.

means that when interacting with the election, all honest voters' ballots might as well contain fake votes, and the adversary would not notice: thus no information on the votes is leaked.

There is however an issue: in many protocols, the Tally algorithm does not only publish the result, but also associated data, *e.g.* zero-knowledge proofs intended to guarantee that the result correctly corresponds to the tally of the bulletin board. As explained above, the adversary always gets to see the result of tallying the real ballots. Hence, showing him a correct tallying proof would always let him distinguish between the two scenarios. Indeed, when he sees fake ballots, the result he sees does not match the tally of the bulletin board he observes, which would become apparent if the adversary had access to the real proofs of correct tallying. To solve this problem, the authors in [26] require that the challenger in the hb-BPRIV game is able to simulate these proofs. That is, there must exist an algorithm $\mathsf{SimProof}(\mathsf{BB}, r)$ that intuitively returns a simulated proof that r is the correct result for BB. This algorithm is used to produce a fake proof to show the adversary when he sees the fake ballots. Note that the SimProof algorithm is not explicitly assumed to produce a convincing fake proof, but if it does not then the hb-BPRIV property will most likely be unsatisfiable. Also note that the SimProof algorithm only has access to public data that the adversary sees, and not to *e.g.* the election private key: this ensures it cannot leak such confidential data to the adversary. Typically, the challenger can implement such a simulator for zero-knowledge proofs by manipulating the random oracle, when considering the random oracle model. We abstract from such details here, and will simply require the existence of the SimProof algorithm, regardless of how it is implemented.

Formally, the hb-BPRIV property is defined as follows.

Definition 46 (hb-BPRIV [26])**.** *Let $\mathcal{V}$ be a voting scheme. Consider the game $\mathsf{Exp}^{\mathsf{hb\text{-}BPRIV},\beta}_{\mathcal{A},\mathcal{V}}$ defined in Figure 6.1. $\mathcal{V}$ satisfies hb-BPRIV if for any polynomial adversary $\mathcal{A}$,*

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{hb\text{-}BPRIV},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{hb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|$$

is negligible in λ.

In [26], the authors introduce two additional properties of voting systems, that are required alongside hb-BPRIV to actually guarantee vote privacy.

Strong consistency. The first one is *strong consistency*. This property can be seen as a stronger version of the consistency assumption we made in the previous chapter. It intuitively states that the result computed by the Tally algorithm on some board BB is always equal to the counting function applied to the votes contained in the ballots in BB. This should hold even if BB has been produced by the adversary, as long as he only proposes ballots that satisfy some validity sub-condition $\mathsf{ValidInd}$, that is a consequence of the actual validity check Valid. Informally, strong consistency ensures that the Tally algorithm itself only computes the counting function, and cannot accept some commands from the adversary encoded in dishonest ballots, that would trigger it *e.g.* to leak unwanted information or consider a bulletin board invalid. Formally, strong consistency is defined as follows.

Definition 47 (Strong consistency [26])**.** *A voting scheme $\mathcal{V}$ is strongly consistent if there exist algorithms* $\mathsf{extract}$ *and* $\mathsf{ValidInd}$ *such that:*

- *For any $id \in \mathcal{I}$, and any vote v, if $(\mathsf{pk}, \mathsf{sk})$ are generated by Setup, U by $\mathsf{Register}$, and (p, b) by $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$, then $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) = (id, v)$ with overwhelming probability.*

- *For any pk produced by Setup, and any $\mathsf{BB}, id, (p, b)$ produced by an adversary,*

$$\mathsf{Valid}(id, (p, b), \mathsf{BB}, \mathsf{pk}) = \top \;\Rightarrow\; \mathsf{ValidInd}(p, b) = \top.$$

- *For any adversary $\mathcal{A}$ that only returns boards BB such that $\mathsf{ValidInd}(p, b) = \top$ for each $(p, b) \in \mathsf{BB}$, the advantage $\mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)$ is negligible, where $\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{A},\mathcal{V}}$ is defined as the following game:*

$\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{A},\mathcal{V}}(\lambda)$

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $\mathsf{U}[id] \leftarrow \mathsf{Register}(id)$
$\mathsf{BB} \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{U})$
$(r, \Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$
if $r \neq \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, p_1, b_1), \ldots, \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p_n, b_n))$
  where $\mathsf{BB} = [(p_1, b_1), \ldots, (p_n, b_n)]$
then return 1
else return 0

Strong correctness. The second companion property to $\mathsf{hb\text{-}BPRIV}$ introduced in [26] is *strong correctness*. This property basically ensures that no matter what ballot the adversary manages to cast to the ballot box, honestly generated ballots can always be accepted afterwards.

Definition 48 (Strong correctness [26])**.** *A voting scheme satisfies* strong correctness *if for any adversary $\mathcal{A}$ the advantage $\mathbb{P}[\mathsf{Exp}^{\mathsf{SCor}}_{\mathcal{A},\mathcal{V}}(\lambda) = 1]$ is negligible, where $\mathsf{Exp}^{\mathsf{SCor}}_{\mathcal{A},\mathcal{V}}$ is defined by*

$\mathsf{Exp}^{\mathsf{SCor}}_{A,\mathcal{V}}(\lambda)$

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $\mathsf{U}[id] \leftarrow \mathsf{Register}(id)$
$(\mathsf{BB}, id, v) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{U})$
$(p, b) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$
if $\mathsf{Valid}(id, (p, b), \mathsf{BB}, \mathsf{pk}) \neq \top$
then return 1
else return 0

Ideal functionality. The authors of [26] show that if a voting scheme is hb-BPRIV, strongly consistent and strongly correct, then it satisfies a simulation-based notion of vote privacy. Informally, assuming an honest ballot box, the scheme then securely implements an ideal functionality, that describes a system where honest voters secretly send their votes to the functionality. In this ideal system, the adversary can only add his own dishonest votes and see the result as computed by the counting function, but he has no way of learning the honest voters' votes.

We will formally define such simulation-based properties in the following sections, as we prove that our mb-BPRIV notion implies stronger variants of this property.

Missing assumption. While designing our definition, it came to our attention that the authors of [26] missed an assumption that is necessary for their proof that hb-BPRIV implies simulation-based security. This property is a *correct authentication* notion, that requires that the protocol correctly ensures that voters are authenticated when they cast their votes. We capture this by requiring that the Valid algorithm that is run before adding a ballot to the ballot box ensures that the identity used to cast the ballot is the same as the one that can be extracted from it, as defined below.

Definition 49 (Correct authentication)**.** *A voting scheme has the* correct authentication *property if* $\mathbb{P}[\mathsf{Exp}^{\mathsf{auth}}_{A,\mathcal{V}}(\lambda) = 1]$ *is negligible for any adversary* A*, where*

$\mathsf{Exp}^{\mathsf{auth}}_{A,\mathcal{V}}(\lambda) =$
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(\lambda)$
  $(\mathsf{BB}, id, (p, b)) \leftarrow A^{\mathcal{O}_{\mathsf{reg}}, \mathcal{O}_{\mathsf{corr}}}(\mathsf{pk})$
  $(id', v) \leftarrow \mathsf{extract}(p, b, \mathsf{sk}, \mathsf{U})$
  If $\mathsf{Valid}(id, b, \mathsf{BB}, \mathsf{U}, \mathsf{pk}) \ \wedge \ id' \neq id$
    then return 1 *else return* 0.

Indeed, in a scheme that does not guarantee this property, the adversary might be able to cast ballots in the name of honest voters. This would intuitively let him break the verifiability property, and, following the same ideas as in the previous chapters, the privacy property. Consider for instance a scheme where voters send ballots formed of their encrypted vote together with their identity in clear, but the server does not ensure that the ballot contains the correct identity. Assume a "last vote counts" revote policy is applied, based on the identities in ballots. For instance, in an election with two honest voters Alice and Bob, the tally of BB =

$[(Alice, 0), (Bob, 1), (Alice, 1)]$ would be $\{\!|1, 1|\!\}$, as only the last vote of Alice is counted. Note that, as there is no correct authentication, the second ballot $(Alice, 1)$ could very well have been cast by the adversary. Such a scheme should therefore intuitively not be deemed private. Indeed, if the adversary wishes to learn Bob's vote, he could simply cast the ballot $(Alice, 1)$ (in the name of a dishonest voter he controls), and replace Alice's real vote with 1, as shown above. The result then immediately lets him know that Bob voted 1.

However, this very simple attack is not caught by the original hb-BPRIV. Indeed, an adversary trying to perform it would *e.g.* call $\mathcal{O}\mathsf{voteLR}(Alice, 0, 1)$ and $\mathcal{O}\mathsf{voteLR}(Bob, 1, 0)$, to make honest voters Alice and Bob vote. The adversary would then cast the ballot intended to replace Alice's: $\mathcal{O}\mathsf{cast}(Charlie, (Alice, 1))$ (where Charlie is a dishonest voter). As before, since no authentication check is performed, this ballot is accepted. The board that is tallied, both on the left and on the right (*i.e.* when $\beta = 0$ and $\beta = 1$) is then $[(Alice, 0), (Bob, 1), (Alice, 1)]$, and the associated result is thus $\{\!|1, 1|\!\}$ on both sides. This does not let the adversary distinguish between both sides. This attack can only be captured by requiring that the protocol has the correct authentication property, which is clearly not the case here.

We proved that this additional assumption indeed fixes the problem, *i.e.* that it is sufficient (along with strong consistency and correctness) to ensure that hb-BPRIV implies simulation-based security. A corresponding technical report is currently being written. We will not detail this any more here, as it is not relevant to the case we consider, of a dishonest ballot box. Indeed, the correct authentication property only makes sense if the ballot box is trusted: we cannot require that an untrusted ballot box still properly authenticates voters. Hence, as we will see in the next sections, we do not make this assumption when studying our mb-BPRIV notion.

The next subsection presents how we adapt our model of voting systems to the case of a dishonest ballot box.

6.2.3 Model

We consider the same computational model for voting protocols as in the previous chapter, except for a few extensions. Following the insight on the links between privacy and verifiability from the previous chapters, we will incorporate the verification mechanisms of the protocols in our privacy definition. Indeed, as we pointed out previously, when considering a dishonest ballot box, not modelling these mechanisms defeats any hope of proving a voting scheme private: if no verification at all is performed, the adversary could simply remove all ballots but one from the ballot box, and the result would let him learn the content of that one ballot. From now on, we thus extend our model with a Verify algorithm modelling the verification steps voters should perform. To do that, the voter may need some knowledge they obtained when creating the ballot (*e.g.* the ballot itself, a verification code, ...). Thus the Vote algorithm now returns a state modelling this knowledge. We will however not assume that all voters actually perform the verifications: such an assumption would be unrealistic in general. Instead, in our model, an arbitrary, fixed subset $\mathsf{H}_{\mathsf{check}}$ of the honest voters will perform them, while the others $\mathsf{H}_{\overline{\mathsf{check}}}$ will not.

In addition, since we will consider an untrusted ballot box, we have no guarantee that any validity condition will be checked on ballots before them being added to the bulletin board. Thus, there is no longer a point to consider a $\mathsf{Valid}(id, b, \mathsf{BB}, \mathsf{pk})$ algorithm that, as before, determines whether id can validly cast a ballot b in BB. Instead, we will from now on simply assume a $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk})$ algorithm, that is applied to the whole bulletin board BB before tallying,

to check that the board is valid. Potentially, this algorithm could correspond to successively applying Valid to each ballot in BB, but we do not require this, and just leave ValidBoard abstract.

Definition 50 (Voting scheme)**.** *A voting scheme is a collection of seven algorithms:*

$$\mathcal{V} = (\mathsf{Setup}, \mathsf{Register}, \mathsf{Pub}, \mathsf{Vote}, \mathsf{ValidBoard}, \mathsf{Tally}, \mathsf{Verify}).$$

- $\mathsf{Setup}(1^\lambda)$ *computes a pair of* election keys $(\mathsf{pk}, \mathsf{sk})$ *given a security parameter* λ*.*
- $\mathsf{Register}(1^\lambda, id)$ *generates a* private credential c *for voter* id *and stores the correspondence* (id, c) *in a list* U*, used for modelling purposes.*
- $\mathsf{Pub}(c)$ *returns the public credential associated with a credential* c*.*
- $\mathsf{Vote}(\mathsf{pk}, id, c, v)$ *constructs a ballot* (p, b) *for user* id *with private credential* c*, containing vote* v*, using the public election key* pk*. It also returns a state to the voter, that models what a voter should record,* e.g. *her ballot. One can think about this state as any information a voter would need to record,* e.g. *to verify if the ballot has been cast.*
- $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk})$ *checks that the board* BB *is valid.*
- $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$ *uses the board* BB *and the secret election key* sk *to compute the result of the election, and potentially proofs of good tallying.*
- $\mathsf{Verify}(id, state, \mathsf{BB})$ *represents the checks a voter* id*, with local state* $state$*, should perform on a board* BB *to ensure her vote is counted.*

With this new, extended model for voting protocols, we can now formalise our privacy notion.

6.3 Game-based Security against a Dishonest Ballot Box: mb-BPRIV

In this section we detail two security notions for voting schemes. Ballot privacy ensures that ballots do not reveal information about the underlying votes. Strong consistency guarantees that the result calculated by the tally does not reveal any additional information beyond what is revealed by the counting function which the scheme should implement. We start with the latter.

6.3.1 Strong consistency

As briefly explained in the previous section, strong consistency demands that the tallying process behaves as expected, *i.e.* it returns the result of tallying the votes which underly the ballots on any given board, even a *dishonestly produced* one. In particular, strong consistency excludes insecure tally functions that would *e.g.* remove the first ballot if it corresponds to a vote for candidate A, hence breaking privacy of the first voter.

We slightly adapt the notion introduced by Bernhard et al [26], and presented in Section 6.2. As before, the property considers an adversary who is given the public key for the election, as well as public information for a set $\mathcal{I}$ of registered users. The adversary returns an arbitrary bulletin board. The definition uses an efficient algorithm which from any given ballot can extract a vote and an identity. Strong consistency requires that tallying the board returns the same result as running the desired counting function on the votes underlying the ballots on the board.

However, since in our model we no longer consider that validity checks are performed on ballots before they are cast, the strong consistency property no longer imposes any conditions regarding the validity algorithm. The next definition formalises these ideas.

Definition 51 (Strong consistency)**.** *A voting scheme $\mathcal{V}$ is strongly consistent if there exist two algorithms* $\mathsf{extract}_{\mathsf{id}}$, $\mathsf{extract}_{\mathsf{v}}$ *such that:*

- *For any $id \in \mathcal{I}$, and any vote v, if* $(\mathsf{pk}, \mathsf{sk})$ *are generated by* Setup, U *by* Register, *and* (p, b) *by* $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$, *then* $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) = id$ *and* $\mathsf{extract}_{\mathsf{v}}(\mathsf{sk}, b) = v$ *with overwhelming probability. We then write* $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) = (\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p), \mathsf{extract}_{\mathsf{v}}(\mathsf{sk}, b))$ *the extraction function that extracts (id, v) from (p, b).*
- *For any adversary $\mathcal{A}$, the advantage* $\mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)$ *is negligible, where* $\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{A},\mathcal{V}}$ *is defined as the following game:*

$\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{A},\mathcal{V}}(\lambda)$

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $\mathsf{U}[id] \leftarrow \mathsf{Register}(id)$
$\mathsf{BB} \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{U})$
$(r, \Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$
if $r \neq \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, p_1, b_1), \ldots, \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p_n, b_n))$
  where $\mathsf{BB} = [(p_1, b_1), \ldots, (p_n, b_n)]$
then return 1
else return 0

Notation: for any board $\mathsf{BB} = [(p_1, b_1), \ldots, (p_n, b_n)]$, any sk, U and extract, we denote $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})$ the list of extractions of the ballots in BB, *i.e.*

$$[\mathsf{extract}(\mathsf{sk}, \mathsf{U}, p_1, b_1), \ldots, \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p_n, b_n)].$$

We also write $\overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})$ the list obtained by removing all $\perp$ elements from $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})$. Note that, by definition of a counting function (Definition 45), we have

$$\rho(\overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})) = \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})).$$

6.3.2 mb-BPRIV

We start with a high level discussion of the notion which we introduce in this section, discuss its salient features over hb-BPRIV, and then provide a formal definition.

We consider a game which pits an adversary $\mathcal{A}$ against a voting scheme. Just as in hb-BPRIV, the adversary has partial information about honest users' votes: for each such user he selects a left-or-right challenge consisting of two votes v_0 and v_1. The game computes the ballots corresponding to the votes but returns to the adversary the ballot which corresponds to v_β, for some hidden bit β which the adversary needs to determine. The game keeps track of both BB_0 and BB_1 (the ordered list of ballots calculated in response to the adversary's queries) – the adversary sees, essentially, BB_β.

The adversary then creates a public bulletin board BB, by using the honest votes and arbitrary other votes it creates (potentially, using the voting credentials of the set of corrupt voters). This

models the adversary being in control of the ballot box, and is a crucial difference with hb-BPRIV: $\mathcal{A}$ can now manipulate the contents of the ballot box in any way he wants, by submitting any sequence of ballots he can construct, rather than only being allowed to add his own ballots to it.

Following the insights from our result on privacy and verifiability from the previous chapters, we now model the verification steps of the protocol. Before we get to the tallying phase, voters in $\mathsf{H}_{\mathsf{check}}$ will perform these steps, which are intuitively intended to check that their votes are indeed represented in the board BB provided by the adversary. If some of these verifications fail, the election is called off and no tally is computed. This will turns out to be most important: since the adversary now controls the ballot box, he could potentially manipulate its content, *e.g.* removing ballots. These verifications mean that if he wants to see the result, he has to be careful not to upset the voters in $\mathsf{H}_{\mathsf{check}}$. As we will see later on, this intuitively introduces a distinction between the privacy level that voters who check and do not check get. Of course, $\mathsf{H}_{\mathsf{check}}$ can be set to the empty set if we do not want to consider any verification steps. Provided all voters in $\mathsf{H}_{\mathsf{check}}$ are satisfied with BB, the game then proceeds to the tallying phase.

The key aspect of the definition is how the game computes the tally it returns to the adversary. When the adversary is in the "real" world where he sees BB_0 the game simply tallies BB (as in hb-BPRIV). The case of the "fake" world where he sees BB_1 is much more complex. In hb-BPRIV, the intuition was that he should still see the tally from the "real" board. However, there is no longer a real and a fake board (as was the case in hb-BPRIV): the only board we have is the one produced by the adversary.

Therefore, our key idea is that we need to determine *how* the adversary manipulated the votes on BB_1 to produce the board he returned to be tallied. That is, we need to somehow (efficiently) determine which of the honest ballots have been cast, on which position on the bulletin board, and which ones have been removed. Once this transformation is determined, the game applies it to BB_0, tallies the resulting board and returns the result to the adversary. We explain a bit later how an insecure scheme would allow a distinguishing attack in the game we outlined above.

Technically, we represent the transformation which describes how the adversary constructs BB from BB_1 as a *selection function*, and we formalise the process of recovering this transformation as a *recovery* algorithm.

Definition 52 (Selection function)**.** *For $m, n \geq 1$, a* selection function for m, n *is any mapping*

$$\pi : [\![1, n]\!] \longrightarrow [\![1, m]\!] \cup (\{0,1\}^* \times \{0,1\}^*)$$

Intuitively, π represents the process used by an adversary to construct a bulletin board BB of n ballots from a given board BB_1 of m ballots. For $i \in [\![1, n]\!]$, $\pi(i)$ indicates how to construct $\mathsf{BB}[i]$:

- $\pi(i) = j \in [\![1, m]\!]$ means this element is the jth from BB_1;
- $\pi(i) = (p, b)$ means that this element is (p, b).

A bit more formally:

Definition 53 (Applying a selection function to a board)**.** *Consider a selection function π for $m, n \geq 1$. The function $\overline{\pi}$ associated to π maps an extended board BB_0 of length m to a board $\overline{\pi}(\mathsf{BB}_0)$ of length n such that for any $j \in [\![1, n]\!]$,*

$$\overline{\pi}(\mathsf{BB}_0)[j] = \begin{cases} (p, b) & \text{if } \pi(j) = i \text{ and } \mathsf{BB}_0[i] = (id, (p, b)) \\ (p, b) & \text{if } \pi(j) = (p, b) \end{cases}.$$

The "recovery" algorithm which recovers the selection function used by the adversary takes as input two boards and some additional data d (intuitively, this piece of data contains the link between voter identities and public credentials).

Definition 54 (Recovery algorithm)**.** *We call* recovery algorithm *any algorithm* Recover *that, given a board* BB*, an extended board* BB_1*, and some additional data* d *as input, returns a selection function for* $|\mathsf{BB}_1|$*,* n *for some* n*.*

We discuss the role of Recover and how it can be interpreted after we provide our formal definition for mb-BPRIV.

The following definition formalises ballot privacy of some scheme $\mathcal{V}$ in a setting where the adversary $\mathcal{A}$ controls the ballot box. Recall that we consider some fixed set of voter identities $\mathcal{I}$ partitioned between two sets H and D of honest and dishonest voters. We also assume that some fixed subset $\mathsf{H}_{\mathsf{check}}$ of H of users perform whatever checks the scheme expects to be executed before the tally is performed. The execution described in Figure 6.2 starts with the generation of a public key for the election and its associated secret key (to be used for tallying). Next, a number of voters from an arbitrary set $\mathcal{I}$ are registered. As before, we keep this aspect of the execution fairly abstract: we assume the registration algorithm/protocol is executed for each user $id \in \mathcal{I}$ and we only record the secret credential c and its associated public credential $\mathsf{Pub}(c)$. We use respectively arrays U and PU to record these. We also use array CU to record the secret credentials for some (arbitrary) set of dishonest users D.

The adversary gets as input the public data pk, PU, and the corrupt credentials CU. It also gets access to a left-right voting oracle. On input an identity id and two potential votes v_0 and v_1, the oracle computes two ballots for id, one for each adversarially selected vote. It records the first ballot in list BB_0 and the second in list BB_1. Recall that we model a voting algorithm which is stateful. This is a necessary feature if one wants, as we do, to consider voters who perform additional actions after they have voted (*e.g.* checking that their ballot has been cast). For each user, we store the resulting state (for both worlds) in arrays V_0 and V_1, respectively. This phase corresponds to the voting phase where the users submit their ballots. Then, the adversary prepares a bulletin board BB which it would like to be tallied. If the bulletin board does not pass the validity test then the tallying does not occur and the adversary needs to output his guess at this point.

Otherwise, the adversary gets control over the users who check via the oracle $\mathcal{O}\mathsf{verify}$, to which it submits arbitrary identities. The oracle records the set of users who have checked in variable $\mathsf{Checked}$ and the set of users for which the check was successful in variable Happy.

To force the adversary to trigger verifications for all of the voters who should check, the game halts without letting $\mathcal{A}$ output his guess if $\mathsf{Checked}$ does not contain $\mathsf{H}_{\mathsf{check}}$.

If all of the voters who should check do check successfully, then the adversary gets to see the tally of the election. Otherwise, *i.e.* if some voters are not satisfied with their verifications, the election is called off, *i.e.* the adversary must produce his guess without seeing the tally.

Finally, a crucial aspect of our definition is how the experiment calculates the tally. In the real execution (*i.e.* $\beta = 0$) the tally is simply executed on BB. In the fake execution (*i.e.* $\beta = 1$) the tally first employs the Recover algorithm which parametrises the game to determine how the adversary has tampered with the votes it has seen (*i.e.* BB_1) to produce the board it asks to be tallied. We explain later precisely how the choice of the Recover algorithm influences the security level guaranteed by this definition. Then the game applies the transformation obtained this way to BB_0. The resulting board is tallied and the result, together with a simulated proof (as in hb-BPRIV), is returned to the adversary.

$\mathsf{Exp}^{\text{mb-BPRIV},\textsc{Recover},\beta}_{\mathcal{A},\mathcal{V}}(\lambda)$

$\mathsf{V}_0, \mathsf{V}_1, \mathsf{Checked}, \mathsf{Happy} \leftarrow \emptyset$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $c \leftarrow \mathsf{Register}(1^\lambda, id)$ $\mathsf{U}[id] \leftarrow c, \mathsf{PU}[id] \leftarrow \mathsf{Pub}(c)$
for all $id \in \mathsf{D}$ **do** $\mathsf{CU}[id] \leftarrow \mathsf{U}[id]$
$\mathsf{BB} \leftarrow \mathcal{A}^{\mathcal{O}\mathsf{voteLR}}(\mathsf{pk}, \mathsf{CU}, \mathsf{PU})$
if $\mathsf{H}_{\mathsf{check}} \not\subseteq \mathsf{V}_0, \mathsf{V}_1$ **then return** $\bot$;
if $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \bot$ **then** $d \leftarrow \mathcal{A}()$; **output** d
$\mathcal{A}^{\mathcal{O}\mathsf{verify}_{\mathsf{BB}}}()$
if $\mathsf{H}_{\mathsf{check}} \not\subseteq \mathsf{Checked}$ **then return** $\bot$
if $\mathsf{H}_{\mathsf{check}} \not\subseteq \mathsf{Happy}$ **then** $d \leftarrow \mathcal{A}()$;
if $\mathsf{H}_{\mathsf{check}} \subseteq \mathsf{Happy}$ **then** $d \leftarrow \mathcal{A}^{\mathcal{O}\mathsf{tally}_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}}()$
output d.

$\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$

if $id \notin \mathsf{H}$ **then return** $\bot$
$(p_0, b_0, state_0) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$
$(p_1, b_1, state_1) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_1)$
$\mathsf{V}_0[id] \leftarrow state_0, \mathsf{V}_1[id] \leftarrow state_1$
$\mathsf{BB}_0 \leftarrow \mathsf{BB}_0 \parallel (id, (p_0, b_0))$
$\mathsf{BB}_1 \leftarrow \mathsf{BB}_1 \parallel (id, (p_1, b_1))$
return (p_β, b_β).

$\mathcal{O}\mathsf{verify}_{\mathsf{BB}}(id)$ for $id \in \mathsf{H}_{\mathsf{check}}$

$\mathsf{Checked} \leftarrow \mathsf{Checked} \cup \{id\}$
if $\mathsf{Verify}(id, \mathsf{V}_\beta[id], \mathsf{BB}) = \top$ **then**
  $\mathsf{Happy} \leftarrow \mathsf{Happy} \cup \{id\}$

$\mathcal{O}\mathsf{tally}_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 0$

$(r, \Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$
return (r, Π)

$\mathcal{O}\mathsf{tally}_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 1$

$\pi \leftarrow \textsc{Recover}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$
$\mathsf{BB}' \leftarrow \overline{\pi}(\mathsf{BB}_0)$
$(r, \Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}', \mathsf{sk})$
$\Pi' \leftarrow \mathsf{SimProof}(\mathsf{BB}, r)$
return (r, Π')

Figure 6.2 – The mb-BPRIV game.

Definition 55 (mb-BPRIV *w.r.t.* a recovery algorithm)**.** *Let $\mathcal{V}$ be a voting scheme, and* Recover *a recovery algorithm. Consider the game $\mathsf{Exp}^{\textit{mb-BPRIV},\textsc{Recover},\beta}_{\mathcal{A},\mathcal{V}}$ defined in Figure 6.2. $\mathcal{V}$ satisfies mb-BPRIV* w.r.t. Recover *if for any polynomial adversary $\mathcal{A}$,*

$$|\mathbb{P}(\mathsf{Exp}^{\textit{mb-BPRIV},\textsc{Recover},0}_{\mathcal{A},\mathcal{V}}(\lambda)=1) - \mathbb{P}(\mathsf{Exp}^{\textit{mb-BPRIV},\textsc{Recover},1}_{\mathcal{A},\mathcal{V}}(\lambda)=1)|$$

is negligible in λ.

Our definition is parametrised by a recovery algorithm, which is a rather non-standard feature. We explain the role it plays through an example. One way to think about the recovery algorithm is that it aims to detect the actions which the adversary took when tampering with the board. That is, informally, Recover tries to understand how each ballot on the bulletin board to be tallied has been created, *i.e.* was it submitted by an honest user, was it created by the adversary, was it submitted by an honest user but modified by the adversary, *etc.* We will now describe the role of the recovery algorithm, but let us first emphasise that its presence does *not* restrict the adversary's abilities. The adversary may still perform any (polynomial-time) computation to obtain the bulletin board he submits to the game. Depending on the recovery algorithm we choose, and on the scheme we consider, it may or may not be possible for the adversary to have behaviours against this scheme that the recovery algorithm *cannot detect*.

Our main idea is that, as explained in introduction on the Helios example, depending on the context (trust assumptions, usage scenario,...) we may wish to consider that some of the possible behaviours of the adversary – some ways in which he manages, against the studied scheme, to

tamper with the votes, *e.g.* remove votes, modify them, and so on – should not constitute an attack. We use the recovery algorithm to finely specify which such behaviours are considered an attack against mb-BPRIV, and which are not. More precisely, as we explain later on, for a given recovery algorithm, any behaviour that is correctly detected by it will not be considered an attack against mb-BPRIV, while any behaviour that is not will constitute an attack.

The following example sheds some light on how Recover plays this role. We show how an adversary can win mb-BPRIV when interacting with a voting scheme that fails to prevent him from tampering with the board in a way undetected by Recover.

Assume a voting scheme that is flawed, in the sense that it fails to prevent the adversary from copying the vote underlying a ballot. To win mb-BPRIV against that scheme, the adversary can proceed as follows. First, he submits $(id, 1, 0)$ to the voting oracle. The game calculates b_0 and b_1 and returns b_β to the adversary (for legibility, we omit the public credentials here). The adversary returns for tallying a board containing b_β, b'_β, where the adversary turned b_β into an equivalent ballot b'_β, which contains the same vote as b_β. In the left world, the tally returns 2. What happens in the right world depends on which recovery algorithm we consider. If we pick a recovery algorithm that does not detect that b'_β is a duplicate of b_β, then it interprets b'_β as a fresh adversarial ballot. The board which will be tallied is then b_0, b'_1, so the result would be 1. The result would differ from the left world, and the adversary would win the game. Our flawed scheme would thus be declared insecure for this recovery algorithm: as expected, the behaviour not detected by Recover is considered an attack. Conversely, if we instead pick a recovery that detects that b'_1 is a copy of b_1, then the board submitted to tally would be b_0, b'_0, where b'_0 is obtained from b_0 using the same action the adversary used on b_1, so the result would also be 2, and it would not help the adversary to distinguish. The scheme, although flawed, would then be declared mb-BPRIV for this recovery algorithm, which detects the copying action. That is, the adversarial behaviour made possible by the flaw would not be considered an attack against mb-BPRIV with that recovery algorithm. Intuitively, such a variant of mb-BPRIV corresponds to studying privacy when we already know that copying attacks are feasible, but are considered tolerable. In that case we wish to know whether *other* attacks exist.

We see here that the Recover we choose determines which adversarial behaviours, if they are possible against the scheme considered, constitute an attack against mb-BPRIV. In other words, Recover determines the security level provided by mb-BPRIV. Our example of a scheme that fails to prevent ballots from being copied would be declared insecure for a Recover that does not detect copied ballots, as this Recover is unable to detect what the adversary did. However it would be secure for a Recover that does detect copies. The second Recover detects more possible actions from the adversary, and hence allows the adversary to do more against the scheme without breaking mb-BPRIV. Therefore, this variant of recovery algorithm yields weaker security guarantees.

More generally, only transformations which Recover can detect will be "acceptable" behaviours from the adversary against the scheme, *i.e.* will not be considered attacks. An adversary that manages, against the considered scheme, to perform some actions that Recover does not detect will break mb-BPRIV, as in the example. Thus, proving a given voting scheme mb-BPRIV for a Recover that detects *less* behaviours from the attacker gives *stronger* security guarantees: it means that the scheme prevents any behaviour not detected by Recover, otherwise mb-BPRIV would be violated.

Finally, coming back to our examples, it is instructive to consider the case where the scheme itself correctly detects and discards such ballot copies (to offer better privacy). Then, even if we

pick the first recovery algorithm, the one that does not detect that b'_β is a duplicate of b_β, the scheme would satisfy mb-BPRIV. Indeed, as the scheme itself detects and prevents the adversary from copying ballots, that behaviour will not be possible against the scheme, and cannot be used to break mb-BPRIV, even if RECOVER does not detect this behaviour. Just as intuition should say, such a scheme would be mb-BPRIV with a recovery algorithm that detects less, which ensures a stronger level of security.

In the case of the Helios protocol, the scheme does not prevent an adversary from modifying the votes of voters who do not verify. As explained in introduction, depending on the usage scenario, this may or may not be an issue. Hence we may want it to be considered an attack or not, depending on the context. In a context where this should constitute an attack, we would study the scheme with a recovery algorithm that is unable to detect ballots that have been modified, and simply discards such ballots. If however we want to specifically exclude this behaviour as an attack, we would pick a recovery algorithm that detects this modification, and applies it on the real board. In the first case, a scheme such as Helios would be considered insecure. In contrast, in the second case it could be secure, provided it prevents any adversarial behaviours other than the one we specifically allowed – *e.g.* it still must prevent the adversary from changing the votes of voters who *do* verify. We present examples of such recovery algorithms in the next subsection.

6.3.3 Instantiations of mb-BPRIV

In this section we describe three instantiations of mb-BPRIV with a few relevant recovery algorithms, that will later on be used for the schemes of our case study. Recall that the recovery algorithm aims to determine how the adversary tampered with the board. For clarity, in our examples we indicate in the indices of the recovery algorithms the actions which we expect each recovery algorithm to be able to detect. For example, $\text{RECOVER}^{\mathsf{del},\mathsf{reorder}}$ would be expected to detect, for each vote in turn, if the adversary has blocked it from appearing in the final tally, or if it has changed the order in which it was cast. We detail this recovery algorithm and discuss how it works. We then provide two variations: one which detects adds an additional class of adversarial behaviours (and thus makes the associated mb-BPRIV variant weaker) and one which restricts the power of the recovery algorithm, detecting fewer adversarial behaviours (and thus makes the associated variant stronger).

Recovery that detects del + reorder

We start with a recovery algorithm that detects adversarial behaviours where the adversary changes the order of the votes in the ballot box, and removes the votes of the voters who do not run the verification algorithm. However this recovery algorithm does not accept the adversary replacing these votes with other votes of his own choosing. In other words, analysing a scheme with mb-BPRIV using this recovery algorithm means that we consider that even if the scheme does not prevent the adversary from reordering and removing votes, this does not constitute not an attack. However, the scheme not preventing the adversary from *changing* votes will be.

Below, we informally describe the transformation the recovery algorithm recovers (as it is applied to the board in the right world).

Given BB_1 and BB, when applied to BB_0, $\text{RECOVER}^{\mathsf{del},\mathsf{reorder}}$ will construct a board BB' where

- Each ballot in BB that comes from BB_1 is replaced with the ballot at the same position in BB_0.

$\textsc{Recover}_{\mathsf{U}}^{\mathsf{del,reorder}}(\mathsf{BB}_1, \mathsf{BB})$

$L \leftarrow []$;
for $(p, b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $L \leftarrow L \parallel j$ (in case several such j exist, pick the first one)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ **then**
    $L \leftarrow L \parallel (p, b)$
$L' \leftarrow [i | \mathsf{BB}_1[i] = (id, (p, b)) \ \wedge \ id \in \mathsf{H}_{\mathsf{check}} \ \wedge \ (p, b) \notin \mathsf{BB}]$
$L'' \leftarrow L \parallel L'$
return $(\lambda i.\ L''[i])$

Figure 6.3 – The $\textsc{Recover}^{\mathsf{del,reorder}}$ algorithm.

- The other ballots in BB are considered to be cast, provided they do not belong to a honest voter, *i.e.* if they do not extract to a honest identity. They are added to BB' as is.
- In addition, all ballots in BB_0 created by honest voters who check their votes are added to BB', regardless of whether these voters' ballots actually occur in BB.

The details of the $\textsc{Recover}^{\mathsf{del,reorder}}$ algorithm are in Figure 6.3.

Recovery that detects del + change + reorder

Here we describe a recovery algorithm that accepts adversarial behaviours where the adversary changes the order of the votes in the ballot box, or removes or changes the votes of the voters who do not verify. The associated variant of mb-BPRIV hence considers that the adversary managing to reorder, remove, or change these votes against the scheme does not constitute an attack.

Intuitively, given BB_1 and BB, when applied to BB_0, algorithm $\textsc{Recover}^{\mathsf{del,reorder,change}}$ will construct a board BB' where

- Each ballot in BB that comes from BB_1 is replaced with the corresponding ballot from BB_0.
- The other ballots in BB are considered to be cast, even if they belong to an honest voter. They are added to BB' as is.
- All the ballots registered for voters who check in BB_0 are added to BB', regardless of whether these voters' ballots actually occur in BB.

Formally, $\textsc{Recover}^{\mathsf{del,reorder,change}}$ is defined in Figure 6.4.

Ideal recovery algorithm

We finally describe a recovery algorithm that, so to speak, does not accept any adversarial behaviour. That is, it will always discard any modification (removing, blocking, reordering...) performed by the adversary on any honest vote, rather than detecting it and applying it to the left board. Studying a voting scheme with this recovery intuitively means that we wish to consider the scheme insecure as soon as it fails to prevent the adversary from tampering in any

$\textsc{Recover}_{\mathsf{U}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\mathsf{BB}_1, \mathsf{BB})$

$L \leftarrow []$;
for $(p, b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $L \leftarrow L \parallel j$ (if several such j exist, pick the first one)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}_{\mathsf{check}}$ **then**
  $L \leftarrow L \parallel (p, b)$
$L' \leftarrow [i | \mathsf{BB}_1[i] = (id, (p, b)) \ \wedge \ id \in \mathsf{H}_{\mathsf{check}} \ \wedge \ (p, b) \notin \mathsf{BB}]$
$L'' \leftarrow L \parallel L'$
return $(\lambda i.\ L''[i])$

Figure 6.4 – The $\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$ algorithm.

way with the honest votes – even of voters who do not verify. Of course, such a strong notion of security will typically be impossible to achieve unless we consider a scenario where every honest voter verifies.

Intuitively, given BB_1 and BB, when applied to BB_0, $\textsc{Recover}^{\emptyset}$ will construct a board BB' where

- Each ballot in BB_0 is added to BB', in the same order, regardless of whether the corresponding ballot from BB_1 actually occurs in BB.
- The other ballots in BB that belong to a dishonest voter are considered to be cast, and are added to BB' as is.

Formally, the $\textsc{Recover}^{\emptyset}$ algorithm is displayed in Figure 6.5.

$\textsc{Recover}_{\mathsf{U}}^{\emptyset}(\mathsf{BB}_1, \mathsf{BB})$

$L \leftarrow [1, \ldots, |\mathsf{BB}_1|]$;
for $(p, b) \in \mathsf{BB}$ **do**
  if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ **then** $L \leftarrow L \parallel (p, b)$
return $(\lambda i.\ L[i])$

Figure 6.5 – The $\textsc{Recover}^{\emptyset}$ algorithm.

We have stated several times by now that the class of transformations a recovery algorithm can detect relates to the level of security provided by the protocol. To express formally this notion of "level of security", we introduce in the next section a family of simulation-based notions of privacy, which specify precisely what power an adversary has against a voting system.

6.4 Simulation-based Security

In this section we introduce simulation based definitions for the security of voting systems. As usual, we describe a real and an ideal execution scenario for the protocol. The definitions are fairly standard in terms of the underlying communication models, and to a large extent in terms of the ideal functionalities we consider. A major departure from functionalities used in the

literature, *e.g.* [81, 26], is that our ideal functionalities explicitly allow the adversary to influence the list of votes to be tallied, so that they can be implemented by voting schemes where such behaviours are known to be possible, especially when the ballot box is untrusted.

6.4.1 Real execution

We describe the real execution of the protocol in a hybrid model where the protocol is implemented using ideal functionalities for registration and for tallying. As for our game based approach, some parameters are fixed. These include the sets H and D of honest and corrupt voters and the set $\mathsf{H}_{\mathsf{check}}$ of voters who check. All these parameters are assumed hardwired in the algorithms defining the execution. We illustrate in Figure 6.6 the execution setting. It comprises:

- The environment $\mathcal{E}$ is in charge of deciding on the execution phases of the protocol (setup, vote, tally); the environment also decides on the votes of the honest users.
- The adversary $\mathcal{A}$: it receives the ballots of the honest users, controls the corrupt users and produces the bulletin board to be tallied. The adversary is controlled by the environment via a direct communication channel.
- An entity $\mathcal{H}$ models the honest parties in the system. In particular it models the honest voters (for simplicity we do not consider separate entities for each individual voter in the system) and the generation of keys for the election.
- The functionality for registration $\mathcal{R}$, in charge of generating and distributing credentials to voters.
- The functionality for tallying $\mathcal{T}$.

The execution consists of three phases (a setup phase, a voting phase, and a tallying phase). The environment $\mathcal{E}$ sends commands to $\mathcal{H}$ to trigger these phases. $\mathcal{H}$ then informs the other entities of the phase change.

Setup phase. During the setup phase, $\mathcal{H}$ runs the Setup algorithm to generate the election keys $(\mathsf{pk}, \mathsf{sk})$, sends pk to the other entities, and sk to $\mathcal{T}$. $\mathcal{H}$ also asks $\mathcal{R}$ to generate credentials. $\mathcal{R}$ runs the $\mathsf{Register}$ algorithm to generate credentials for each voter. It returns the secret credentials of all voters to $\mathcal{H}$, who forwards the secret credentials of voters in D to $\mathcal{A}$. It also sends the list of public credentials (computed with Pub) to all other entities (not represented in Figure 6.6). Once this is done, $\mathcal{H}$ returns the control to $\mathcal{E}$.

Voting and checking phase. During the voting phase, $\mathcal{E}$ may send any number of $\mathtt{vote}(id, v)$ to $\mathcal{H}$, for honest voters $id \in \mathsf{H}$. When receiving such a command, $\mathcal{H}$ runs the Vote algorithm to obtain a ballot for id containing v, it records the state returned by the voting algorithm and sends id and the ballot to $\mathcal{A}$. Voters in $\mathsf{H}_{\mathsf{check}}$ are supposed to perform some verifications later on, which only makes sense if they have cast a vote. Hence, we only consider environments $\mathcal{E}$ that sends at least one $\mathtt{vote}(id, v)$ command for each $id \in \mathsf{H}_{\mathsf{check}}$. At some point, $\mathcal{E}$ notifies $\mathcal{H}$ that the voting phase is done. At that point, $\mathcal{H}$ asks $\mathcal{A}$ to provide a board BB. $\mathcal{H}$ checks that $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$, and continues the execution. If the check fails, it informs $\mathcal{E}$ that no result will be published. $\mathcal{H}$ then performs the verifications of honest voters. It asks $\mathcal{A}$ in which order the voters should verify. $\mathcal{H}$ then runs the Verify algorithm on BB for each voter in $\mathsf{H}_{\mathsf{check}}$, in the order specified by $\mathcal{A}$. If all of these checks succeed, it continues the execution. Otherwise, it informs $\mathcal{E}$ that no result will be published.

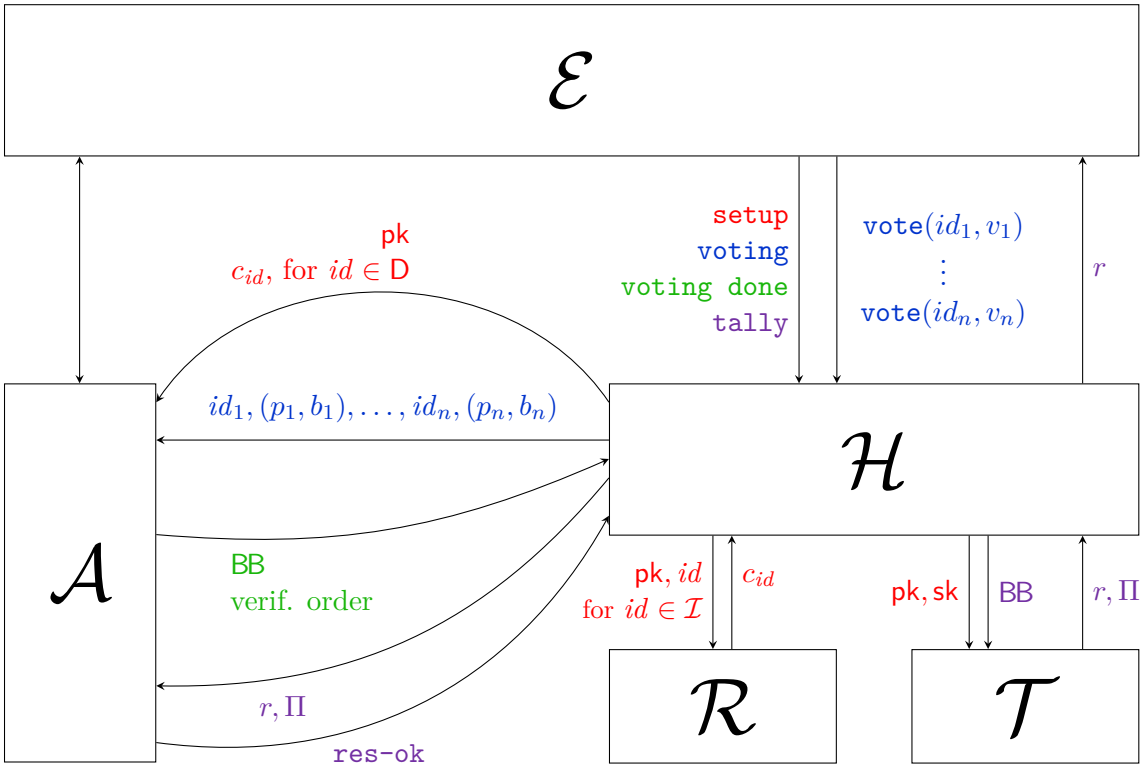


Figure 6.6 – The real execution
with $id_1, \ldots, id_n \in$ H

Tallying phase. During the tallying phase, $\mathcal{H}$ sends BB to $\mathcal{T}$ and asks for the result. $\mathcal{T}$ runs the Tally algorithm, to compute a result (r, Π), and sends it back to $\mathcal{H}$. $\mathcal{H}$ forwards this result to $\mathcal{A}$, asking if the result should be published. Depending on $\mathcal{A}$'s decision, $\mathcal{H}$ sends $\mathcal{E}$ either r or a message informing $\mathcal{E}$ that no result is published. Finally, the environment $\mathcal{E}$ outputs a bit β which serves as the output of realexec$(\mathcal{E}||\mathcal{A}||\mathcal{V})$.

6.4.2 Ideal voting functionality and ideal execution

The ideal execution replaces the honest participants and the functionalities for registration and tallying with a single idealised functionality $\mathcal{F}_{\mathsf{v}}$. The resulting structure of the system is illustrated in Figure 6.7. It comprises

- The environment $\mathcal{E}$: as in the real execution the environment decides changes between the different phases of the execution, decides on the votes of the honest parties, and communicates with the ideal world adversary. As in the real case, we will only consider environments that choose to make each voter in $\mathsf{H}_{\mathsf{check}}$ vote at least once.

- The ideal world adversary $\mathcal{S}$, also called the *simulator*;

- The ideal voting functionality $\mathcal{F}_{\mathsf{v}}$: this component captures the idealised voting scheme. Very roughly, it receives the votes from the honest parties and, when queried, it returns the result of the election. We give a precise description of the voting functionality in the next section.

- The entity $\mathcal{H}'$ is a dummy interface between the environment and the voting functionality (*i.e.* it only forwards the messages between $\mathcal{E}$ and $\mathcal{F}_{\mathsf{V}}$).

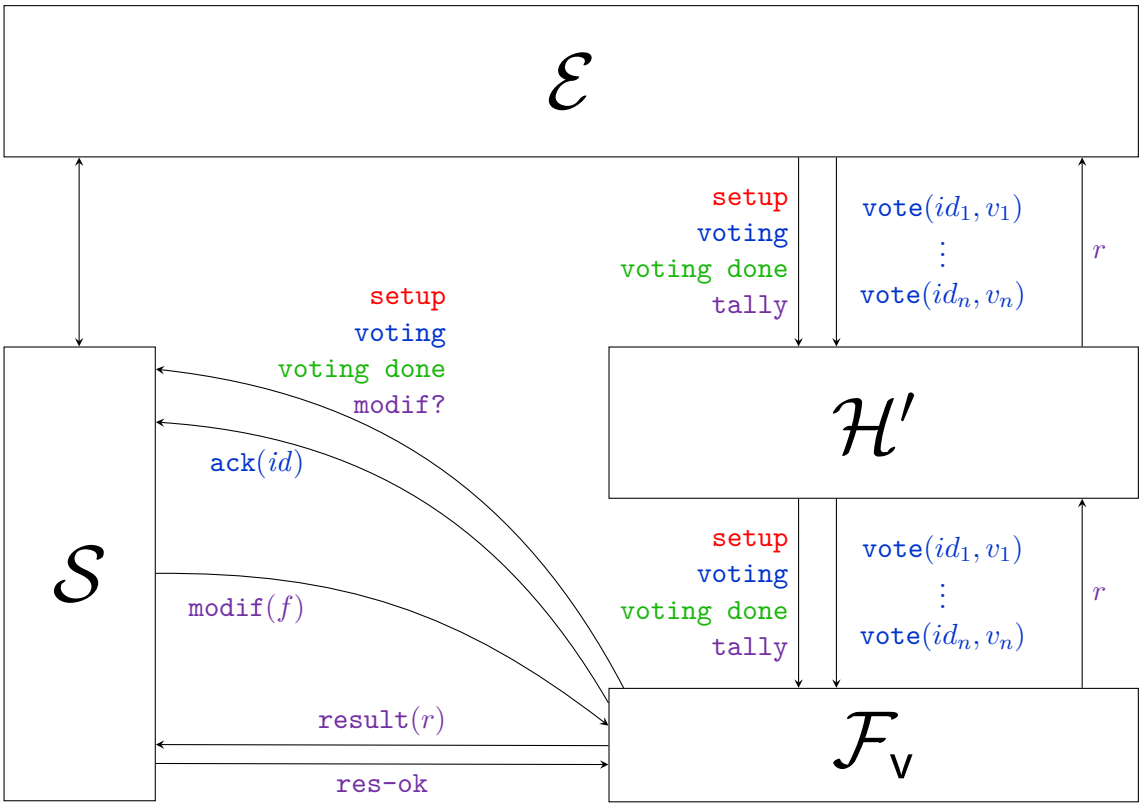


Figure 6.7 – The ideal execution
with $id_1, \ldots, id_n \in \mathsf{H}$

As in the real execution the environment decides when to switch between the three phases of the execution (setup, vote and check, tally) and decides on the votes of the honest parties via messages it sends to $\mathcal{H}'$. In this world $\mathcal{H}'$ is simply a forwarding channel between the environment and the ideal functionality (we explain below how the functionality operates). At some point the environment outputs a bit β which is also the output of $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{V}})$.

Next we describe the ideal functionality which is the key component of the execution, and which encapsulates the level of security guaranteed.

Ideal voting functionality. We consider several ideal functionalities which share the same basic design idea: they collect the votes of the honest parties (in a way which hides them from the adversary). Nonetheless, since we treat the setting where the adversary controls the bulletin board, and can therefore influence what is being tallied, our functionalities reflect this ability. The difference between the functionalities we consider is reflected in how permissive they are with respect to this step.

The functionality $\mathcal{F}_{\mathsf{V}}^{\mathsf{del},\mathsf{reorder}}(\rho)$ is displayed in Figure 6.8. In brief, 1. it ensures that an adversary only learns who voted, and learns the result of the election, computed using ρ, but not what the votes were; 2. it ensures that the votes of honest voters who verify are not removed (though they may be reordered); 3. it allows an adversary to delete the votes of voters who do not verify, but not to change them.

Technically, the functionality maintains a list L of votes submitted by honest voters. Once the voting phase is over, it allows the ideal world adversary $\mathcal{S}$ to submit a *vote tampering function*, *i.e.* a function that describes how $\mathcal{S}$ wishes to manipulate the votes in L, and needs to satisfy a couple of restrictions. Formally:

Definition 56 (Vote tampering function)**.** *A* vote tampering function *is a function f with domain $[\![1, n]\!]$ for some n, such that for any i, $f(i)$ is either an index j, or a pair (id, v) of an identity and a vote.*

Similarly to selection functions introduced in the previous chapter, f is intended to describe how $\mathcal{S}$ wishes to construct a list of $n = |\mathrm{dom}(f)|$ votes using the votes in L. Basically, $f(i)$ describes how to obtain the ith element in this list: if $f(i) = j$, it should be the jth element from L, and otherwise $f(i) = (id, v)$ and it should be (id, v).

Definition 57 (Applying a tampering function)**.** *Applying a vote tampering function f (with domain $[\![1, n]\!]$) to the list L of honest votes results in list $\overline{f}(L)$ of length n defined by*

$$\forall i \in [\![1, n]\!].\ \overline{f}(L)[i] = \begin{cases} L[j] & \text{if } f(i) = j \text{ is an index} \\ (id, v) & \text{if } f(i) = (id, v) \text{ for some } id, v \end{cases}$$

Note that, in $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}(\rho)$, the function f is applied only if it satisfies the requirements outlined above on how it affects the votes corresponding to the honest voters who check. That is, as specified in Figure 6.8, all votes of voters who check must be kept (although not necessarily in order), and no honest votes can be modified (but they can be simply removed). Otherwise, f is rejected, *i.e.* no result is computed and the functionality returns `no tally` instead.

Next, we define $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho)$, a more permissive functionality for voting schemes. This functionality is similar to the previous $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}(\rho)$ but it allows an adversary to change (and not only delete) the votes of honest voters, as long as they do not verify. Technically, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho)$ accepts the same commands as $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}(\rho)$, and answers them identically, except for the `modif`(f) command. In that case, in $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho)$, the checks performed before computing the result are:

- f keeps the votes of all voters who check (as before):

$$\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall v.\ L[i] = (id, v) \implies \exists j.\ f(j) = i.$$

- no votes from voters who verify are modified by f:

$$\forall i, id, v.\ f(i) = (id, v) \implies id \in D \cup \mathsf{H}_{\overline{\mathsf{check}}}.$$

Note that the simulator is thus allowed by $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho)$ to modify the vote of any honest voter who does not verify.

Finally we define a functionality $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, that gives the strongest security guarantees, as it does not allow the adversary to delete, change, nor reorder the votes of honest voters, even if they do not verify. All the adversary may do is cast votes in the name of dishonest voters. This functionality is similar to the previous two, except it checks a stronger condition before computing the result. More precisely, $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$ is identical to $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}(\rho)$, except that the test performed before computing $\rho(\overline{f}(L))$ on command `modif`(f) is that:

$\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}(\rho)$ accepts the following commands:

- on `setup` from $\mathcal{H}'$: send `setup` to $\mathcal{S}$.
- on `voting` from $\mathcal{H}'$: send `voting` to $\mathcal{S}$.
- on `voting done` from $\mathcal{H}'$: send `voting done` to $\mathcal{S}$.
- on $\texttt{vote}(id, v)$ (for $id \in \mathsf{H}$) from $\mathcal{H}'$:
  $L \leftarrow L \parallel (id, v)$; send $\texttt{ack}(id)$ to $\mathcal{S}$.
- on `tally` from $\mathcal{H}'$:
  send `modif?` to $\mathcal{S}$.
- on $\texttt{modif}(f)$ from $\mathcal{S}$: (only once, after `tally`)
  - if f keeps the votes of all voters who check:

    $$\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall v.\ L[i] = (id, v) \implies \exists j.\ f(j) = i.$$

  - and if no honest votes are modified by f:

    $$\forall i, id, v.\ f(i) = (id, v) \implies id \in D.$$

  Then let $r = \rho(\overline{f}(L))$ else let $r =$ `no tally`.
  Send $\texttt{result}(r)$ to $\mathcal{S}$.
- on `res-ok` from $\mathcal{S}$: (only once, after `modif`)
  send r to $\mathcal{H}'$.
- on `res-block` from $\mathcal{S}$: (only once, after `modif`)
  send `no tally` to $\mathcal{H}'$.

Figure 6.8 – The ideal functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}(\rho)$.

- f keeps the votes of all honest voters in the same order:

$$[f(j), j = 1 \ldots |\mathrm{dom}(f)| \mid f(j) \in \mathbb{N}] = [1, \ldots, |L|]$$

- and no honest votes are modified by f:

$$\forall i, id, v.\ f(i) = (id, v) \implies id \in \mathsf{D}.$$

This functionality enforces that no honest votes can be deleted or even reordered. Intuitively, in the presence of a malicious ballot box, this level of security can be guaranteed only if we assume all honest voters verify their votes.

6.4.3 Simulation

As is usual for simulation-based notions, we define security by demanding that environments cannot distinguish between the interaction with the real protocol, or with the ideal functionality

(together with some simulator). However, as the motivating examples from the introduction show, the level of security guaranteed depends on the fact that (some) voters (*i.e.* those in $\mathsf{H}_{\mathsf{check}}$) check that their vote had been cast. Our security definition captures this by considering certain restrictions. Specifically, as mentioned earlier, we will only consider environments who direct all voters in $\mathsf{H}_{\mathsf{check}}$ to cast at least one vote, so that it makes sense for this vote to be verified. We call such an environment *well-behaved*.

Definition 58 (Secure implementation)**.** *We say that a voting scheme $\mathcal{V}$ securely implements an ideal functionality $\mathcal{F}_{\mathsf{v}}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any well-behaved environment $\mathcal{E}$ the distributions of the outputs of* $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$ *and* $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}})$ *are computationally indistinguishable. That is, no probabilistic polynomial-time algorithm should be able to distinguish between these two outputs.*

6.5 mb-BPRIV implies Simulation-based security

Our main technical result, detailed in this section, is that for strongly consistent voting schemes, game-based ballot privacy implies simulation security with respect to a suitable ideal functionality.

Both our game-based security notion and the ideal functionalities are parametrised. The former is parametrised by a recovery algorithm which aims to "detect" how the adversary has tampered with the bulletin board. The latter allows the adversary to submit a tampering function, but only allows certain tampering functions. We show that the two parameters are closely related: mb-BPRIV with respect to some specific recovery algorithm implies simulation-based security, if the tampering function recovered by the recovery algorithm is permitted by the ideal functionality.

6.5.1 Warm-up

To make this relatively complex statement more palatable, we start by writing it in a particular case: we state a warm-up theorem, which establishes this type of relation between the three instantiations of mb-BPRIV from Section 6.3.3 and simulation based security which uses the three ideal functionalities from Section 6.4.2, respectively. Then, we provide a powerful, general theorem which links mb-BPRIV with simulation based security under an abstract assumption on their parameters.

Before we provide our warm-up theorem, we motivate and introduce a mild assumption required by the scheme. All of the recover algorithms considered earlier in this chapter identify the ballots on the board by matching them with the specific calls to the voting oracle which produced them. For this reason, a precondition for the recovery algorithms to work as intended is that distinct calls to the Vote algorithm produce two different ballots (except with negligible probability). We say that a scheme with this property does not produce duplicate ballots. Formally:

Definition 59 (Voting scheme without duplicate ballots)**.** *A voting scheme $\mathcal{V}$ does not produce* duplicate ballots *if for all adversary $\mathcal{A}$, the following probability is negligible in λ.*

$$
\begin{aligned}
\mathbb{P}[\ & (\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda);\\
& \mathsf{U} \leftarrow \mathsf{Register}(1^\lambda,\mathcal{I});\\
& (id, v, id', v') \leftarrow \mathcal{A}(\mathsf{pk},\mathsf{U});\\
& (p,b,s) \leftarrow \mathsf{Vote}(\mathsf{pk},id,\mathsf{U}[id],v); (p',b',s') \leftarrow \mathsf{Vote}(\mathsf{pk},id',\mathsf{U}[id'],v');\\
& (p,b) = (p',b')]
\end{aligned}
$$

$P^{\emptyset}(L, f) = \top$ iff:

- f keeps the votes of all honest voters in the same order:
  $[f(j), j = 1 \dots |\mathrm{dom}(f)| \mid f(j) \in \mathbb{N}] = [1, \dots, |L|]$
- and no honest votes are modified by f:
  $\forall i, id, v.\ f(i) = (id, v) \implies id \in \mathsf{D}$.

$P^{\mathsf{del,reorder}}(L, f) = \top$ iff:

- f keeps the votes of all voters who check:
  $\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall v.$
  $L[i] = (id, v) \implies \exists j.\ f(j) = i$.
- and no honest votes are modified by f:
  $\forall i, id, v.\ f(i) = (id, v) \implies id \in \mathsf{D}$.

$P^{\mathsf{del,reorder,change}}(L, f) = \top$ iff:

- f keeps the votes of all voters who check:
  $\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall v.\ L[i] = (id, v) \implies \exists j.\ f(j) = i$.
- and no votes from voters who check are modified by f:
  $\forall i, id, v.\ f(i) = (id, v) \implies id \in \mathsf{D} \cup \mathsf{H}_{\overline{\mathsf{check}}}$.

Figure 6.9 – Predicates associated with the three functionalities $\mathcal{F}_{\mathsf{v}}^{\emptyset}$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$.

Note that this assumption is only required by the specific Recover algorithms described earlier, but is not necessary in general.

We can now state the following warm-up theorem.

Theorem 10. *Consider a strongly consistent voting scheme $\mathcal{V}$ for counting function ρ which does not produce duplicate ballots. Let* $\mathsf{power} \in \{\emptyset, (\mathsf{del}, \mathsf{reorder}), (\mathsf{del}, \mathsf{reorder}, \mathsf{change})\}$. *If $\mathcal{V}$ satisfies mb-BPRIV with* $\textsc{Recover}^{\mathsf{power}}$, *then $\mathcal{V}$ securely implements* $\mathcal{F}_{\mathsf{v}}^{\mathsf{power}}(\rho)$.

This theorem will be proved later on, as a particular case of our general theorem, that we explain next.

6.5.2 Parametric ideal functionality.

The three ideal voting functionalities from Theorem 10, $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$, and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$, differ only by the check they perform on the tampering function provided by the simulator. They can be seen as instances of a more general functionality, which is parametrised by a predicate P that expresses this check. In other words, P characterises the ability of the simulator to manipulate the votes. The predicate P takes as inputs a list L of pairs (id, v), where id is a voter identity and v is a vote, and a tampering function f. It returns $\top$ or $\bot$, indicating whether the modifications specified by f are allowed on L or not. For instance, in the case of $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$, the predicate $P^{\mathsf{del,reorder}}(f, L)$ checks that f keeps from L the votes of all voters who check, and that it does not modify the votes of any honest voters. If f does not satisfy these two conditions, $P^{\mathsf{del,reorder}}(f, L) = \bot$.

The generalisation is then straightforward: we consider the functionality $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ with the same interface and mostly the same internal behaviour as $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ (Figure 6.8). The only distinction is that the checks performed on f before applying it to L are replaced with a single check that $P(L, f) = \top$.

The three functionalities from Section 6.4 are then instances of the parametric functionality $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$, with the predicates $P^{\emptyset}$, $P^{\mathsf{del,reorder}}$ and $P^{\mathsf{del,reorder,change}}$ displayed in Figure 6.9.

Our generic theorem links mb-BPRIV *w.r.t.* some recovery algorithm Recover with an

ideal functionality which allows tampering satisfying some predicate P whenever the Recover algorithm returns[15] (with overwhelming probability) only tampering functions which satisfy P.

Below, we develop the technical machinery that captures these ideas.

First we relate selection functions (which are the type of functions returned by recovery algorithms) with tampering functions (the functions the simulators provide to the ideal functionalities). This definition can be seen as the analogous definition of applying a selection function to a bulletin board, except that now we operate at vote level (rather than ballot level). In particular, this requires that we recover the votes underlying ballots in the selection function.

Definition 60 (Tampering function associated to a selection function)**.** *Assume a strongly consistent voting scheme $\mathcal{V}$. Let* $(\mathsf{pk}, \mathsf{sk})$ *be a pair of keys generated by the* Setup *algorithm, and* U *be a list of credentials issued by* Register*. Let π be a selection function for two integers m, n. Let then L be the list of length n defined by*

$$\forall i \in [\![1, n]\!].\ L[i] = \begin{cases} \pi(i) & \text{if } \pi(i) \in [\![1, m]\!] \\ \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) & \text{if } \pi(i) = (p, b) \text{ for some } p, b \end{cases}$$

The vote tampering function $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$ *associated to π for* sk *and* U *is the function $\lambda i.\ L'[i]$ (with L' denoting L where $\perp$ elements have been removed).*

As explained above we want to relate mb-BPRIV with some Recover algorithm with an ideal functionalities which put restrictions on how the adversary (the simulator) can tamper with the list of votes collected by the functionality. Which restrictions we can consider depends on the Recover algorithm we choose. As explained previously, the intuition is that if a scheme is mb-BPRIV for a recovery algorithm, then this necessarily entails that the scheme prevents the adversary from performing actions that are not detected by that Recover. Hence, mb-BPRIV will guarantee simulation security *w.r.t.* any ideal functionality that allows such actions to the simulator.

We capture this intuition by the notion of *compatibility*, which we first express on individual selection functions (in practice, produced by Recover). Basically, such a function is *compatible* with the testing predicate associated to the functionality if the associated vote tampering function is allowed by this predicate.

Definition 61 (Selection function compatible with a testing predicate)**.** *Let* $(\mathsf{pk}, \mathsf{sk})$ *be a pair of keys generated by the* Setup *algorithm, and* U *be a list of credentials issued by* Register*. Let P be a predicate, and π be a selection function for m, n. Let L_{id} be a list of ids of length m. We say that π is compatible with P* w.r.t. sk, U*, and L_{id} if for any list L of pairs of the form (id, v) such that $[id \mid (id, v) \in L] = L_{id}$, we have $P(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)) = \top$.*

This notion of compatibility can then be extended from individual selection functions to recovery algorithms which return selection functions. As explained in the paragraph above, the general intuition we want to capture is that Recover is compatible with P if Recover (almost) always returns selection functions compatible with P in normal executions of the scheme (*i.e.* where parameters and ballots are generated honestly).

[15] In this description we overloaded the semantics of the recovery algorithm. Strictly speaking this algorithm returns a selection function, which in turn defines a tampering function.

Definition 62 (Recovery algorithm compatible with a testing predicate)**.** *Let P be a predicate, and* Recover *be a recovery algorithm. We say that* Recover *is compatible with P if for any adversary $\mathcal{A}$, the advantage $\mathbb{P}(\mathsf{Exp}^{\mathsf{comp},P,\textsc{Recover}}_{\mathcal{A},\mathcal{V}}(\lambda)=1)$ is negligible, where $\mathsf{Exp}^{\mathsf{comp},P,\textsc{Recover}}_{\mathcal{A},\mathcal{V}}$ is defined as the following game.*

$\mathsf{Exp}^{\mathsf{comp},P,\textsc{Recover}}_{\mathcal{A},\mathcal{V}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $c \leftarrow \mathsf{Register}(1^\lambda, id)$
  $\mathsf{U}[id] \leftarrow c, \mathsf{PU}[id] \leftarrow \mathsf{Pub}(c)$
for all $id \in \mathsf{D}$ **do** $\mathsf{CU}[id] \leftarrow \mathsf{U}[id]$
$\mathsf{BB} \leftarrow \mathcal{A}^{\mathcal{O}\mathsf{vote}}(\mathsf{pk},\mathsf{CU},\mathsf{PU})$
if $\mathsf{ValidBoard}(\mathsf{BB},\mathsf{pk}) = \bot \vee \mathsf{H}_{\mathsf{check}} \not\subseteq \mathsf{L}_{id}$ **then return** 0
$\pi \leftarrow \textsc{Recover}_{\mathsf{U}}(\mathsf{BB}_0,\mathsf{BB})$
if π *is not compatible with* P w.r.t. $\mathsf{sk},\mathsf{U},\mathsf{L}_{id}$
then return 1 **else return** 0

$\mathcal{O}\mathsf{vote}(id,v)$ *for* $id \in \mathsf{H}$

$(p,b,s) \leftarrow \mathsf{Vote}(\mathsf{pk},id,\mathsf{U}[id],v)$
$\mathsf{BB}_0 \leftarrow \mathsf{BB}_0 \parallel (id,(p,b))$;
$\mathsf{L}_{id} \leftarrow \mathsf{L}_{id} \parallel id$;
return (p,b).

For example, we will show later on that the algorithms $\textsc{Recover}^{\emptyset}$, $\textsc{Recover}^{\mathsf{del},\mathsf{reorder}}$, and $\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$ presented earlier are respectively compatible with $P^{\emptyset}$, $P^{\mathsf{del},\mathsf{reorder}}$, $P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$.

Using the notion of compatibility to relate recovery algorithms and ideal functionality (via their testing predicates), we can now state and prove our general theorem.

6.5.3 General theorem

Our main technical result regarding mb-BPRIV establishes a relation between game-based and simulation-based security for voting, under a minimal compatibility between the parameters of the respective definitions.

Theorem 11 (mb-BPRIV implies simulation)**.** *Let P be a predicate, and* Recover *a recovery algorithm compatible with P. Let $\mathcal{V}$ be a strongly consistent voting scheme for counting function ρ.*

If $\mathcal{V}$ satisfies mb-BPRIV w.r.t. Recover*, then $\mathcal{V}$ securely implements $\mathcal{F}^P_{\mathsf{v}}(\rho)$.*

Before we get to the proof of this result, it is interesting to note that, since we considered arbitrary fixed sets of voters (honest, corrupt, who verify their votes), it means that mb-BPRIV with any such sets of voters entails simulation-based security *with the same sets of voters*. In particular, for an adversary against the scheme that controls n dishonest voters, the secure implementation property guarantees the existence of a simulator (*i.e.* ideal adversary) that only needs to control *the same number* n of dishonest voters. This would not be possible if we had considered a dynamic corruption scenario: the theorem would then only have established the existence of a simulator, that could possibly choose to corrupt more voters than the real adversary does. An interesting question is to study *e.g.* how the proportion of voters who verify among honest voters influences the security of the scheme. Since we relate the real and ideal systems with the same numbers of honest voters, corrupt voters, and voters who verify their vote, it is meaningful to perform this study directly on the ideal system, which we do in a later section.

Proof. Let us first explain the intuition of this result, before detailing its proof. We prove this theorem by constructing a simulator $\mathcal{S}$ that, given black-box access to a real adversary $\mathcal{A}$, ensures that for any well-behaved environment $\mathcal{E}$, the outputs of $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$ and $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P}(\rho))$ are indistinguishable.

The idea is to have $\mathcal{S}$ run $\mathcal{A}$ internally, letting $\mathcal{A}$ communicate with $\mathcal{E}$ through $\mathcal{S}$, and simulate the real execution of the voting scheme to $\mathcal{A}$. To do this, $\mathcal{S}$ generates election keys and credentials on its own, and shows them to $\mathcal{A}$. $\mathcal{S}$ then needs to provide $\mathcal{A}$ with ballots when $\mathcal{E}$ makes voters vote, to obtain a board BB from $\mathcal{A}$. Finally $\mathcal{S}$ must determine which vote tampering function it should submit to $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ to get the tally of BB that $\mathcal{A}$ expects to see.

The issue is that $\mathcal{S}$ does not have access to the actual votes of the voters: $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ only informs $\mathcal{S}$ of who voted, but not of the values of the votes. Hence $\mathcal{S}$ cannot construct ballots containing the real votes to show to $\mathcal{A}$. Instead, $\mathcal{S}$ will construct fake ballots, containing always the same arbitrary fixed fake vote v^*.

After the voting phase is over the adversary returns a bulletin board BB and demands to see the result of the tally. At this point, $\mathcal{S}$ uses the Recover algorithm on the board BB (and its own list of ballots) to determine how $\mathcal{A}$ manipulated the ballots. The recovery algorithm returns a selection function (which encodes how $\mathcal{A}$ has tampered with the honest votes). $\mathcal{S}$ submits the associated vote tampering function to $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$. Notice that from the adversary's point of view the simulation is as in the experiment that defines compatibility of Recover and P (Definition 62). So P, and therefore $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$, will accept the output of Recover. The functionality will return a result to $\mathcal{S}$, who will forward it to the (internally simulated) adversary $\mathcal{A}$, together with a fake proof calculated using the $\mathsf{SimProof}$ algorithm on (r, BB).

Intuitively, when adversary $\mathcal{A}$ is placed in the real execution (*i.e.* is in $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$), its view is as in $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},0}$: the ballots contain the "real" votes for honest parties, and the tally is the tally of the board it provides. When $\mathcal{A}$ is run internally by $\mathcal{S}$ when executing $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P}(\rho))$, the view of the adversary is as in $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},1}$: the ballots contain "fake" votes, and the tally is a fake tally (with a simulated proof). Strong consistency ensures that the result for which $\mathcal{S}$ simulates the proof is exactly the result seen by the adversary in $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},1}$. Note that strong consistency is a separate assumption from $\mathsf{mb\text{-}BPRIV}$: it can be defined independently from $\mathsf{mb\text{-}BPRIV}$, but both properties are needed to entail simulation security.

Since security with respect to $\mathsf{mb\text{-}BPRIV}$ guarantees that the adversary cannot tell the two situations apart, the environment cannot distinguish between $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$ and $\mathsf{idealexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$. This will conclude the proof: this simulator $\mathcal{S}$ can simulate any real adversary $\mathcal{A}$ in the eyes of any well-behaved environment.

A bit more formally, we proceed to construct $\mathcal{S}$ by a succession of game hops, progressively going from $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$ to the ideal execution against $\mathcal{S}$.

Game 0. is just the real execution $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$.

Game 1. is a variant of $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$, where $\mathcal{A}$ does not see ballots for the actual votes chosen by $\mathcal{E}$, but rather fake ballots containing an arbitrary fixed vote v^*. The execution of Game 1 follows that of Game 0, except that

- on $\mathtt{vote}(id, v)$ from $\mathcal{E}$ for some $id \in \mathsf{H}$:
  $\mathcal{H}$ generates two ballots: a real one for v, and a fake one for v^*. That is, if c_{id} is the credential generated for id by $\mathcal{R}$ during the setup phase, $\mathcal{H}$ runs $(p, b, state_{id}) = \mathsf{Vote}(\mathsf{pk}, id, c_i d, v)$

(as in the real execution), and $(p', b', state'_{id}) = \mathsf{Vote}(\mathsf{pk}, id, c_{id}, v^*)$. It records the voter's internal state, and stores $(id, (p, b))$ and $(id, (p', b'))$ respectively in lists BB_0 and BB_1. $\mathcal{H}$ then returns $(id, (p', b'))$ to $\mathcal{A}$ (instead of $(id, (p, b))$ in Game 0).

- on `tally` from $\mathcal{E}$: After obtaining the board BB and the order in which to verify from $\mathcal{A}$, when $\mathcal{H}$ is performing the verification for a voter $id \in \mathsf{H}_{\mathsf{check}}$, it runs $\mathsf{Verify}(id, state'_{id}, \mathsf{BB})$ (instead of $\mathsf{Verify}(id, state_{id}, \mathsf{BB})$ in Game 0).

- During the tallying phase: if all verifications have succeeded, $\mathcal{H}$ does not send BB to $\mathcal{T}$ for tallying, but rather $\mathsf{BB}' = \overline{\pi}(\mathsf{BB}_0)$, where $\pi = \textsc{Recover}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$, and U is the association between identities and secret credentials, that $\mathcal{H}$ knows from the setup phase.

- When $\mathcal{H}$ gets a result (r, Π) from $\mathcal{T}$ in the tallying phase: it simulates a proof $\Pi' = \mathsf{SimProof}(r, \mathsf{BB})$, and sends (r, Π') to $\mathcal{A}$ (instead of (r, Π) in Game 0).

We now show that, for any $\mathcal{A}$ and $\mathcal{E}$, the outputs of Game 0 and Game 1 are indistinguishable, using the assumption that $\mathcal{V}$ is mb-BPRIV. As explained above, the intuition is that Game 0 corresponds to the execution of $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},\beta}_{\mathcal{A},\mathcal{V}}$ when $\beta = 0$, and Game 1 to the execution when $\beta = 1$. Since $\mathcal{V}$ is mb-BPRIV, we are able to show that the outputs of $\mathcal{E}$ in Games 0 and 1 are indistinguishable.

More formally, assume there exists a distinguisher D for the outputs of Game 0 and Game 1. We then construct an adversary $\mathcal{B}$ playing the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},\beta}_{\mathcal{B},\mathcal{V}}$. $\mathcal{B}$ runs $\mathcal{A}$ and $\mathcal{E}$ internally as follows.

- When $\mathcal{B}$ is first allowed to run in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},\beta}_{\mathcal{B},\mathcal{V}}$, it receives pk, CU and PU. It sends pk, CU and PU to $\mathcal{A}$, simulating the setup phase. $\mathcal{B}$ then returns the control to $\mathcal{E}$.

- When $\mathcal{E}$ wishes to send the command `vote`(id, v) to $\mathcal{H}$, $\mathcal{B}$ uses its voting oracle by calling $\mathcal{O}\mathsf{voteLR}(id, v, v^*)$. $\mathcal{B}$ receives a ballot (p, b), and sends $(id, (p, b))$ to $\mathcal{A}$.

- Once $\mathcal{E}$ sends `voting done`, $\mathcal{B}$ asks $\mathcal{A}$ for a board. $\mathcal{A}$ responds with some BB, which $\mathcal{B}$ returns to the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},\beta}_{\mathcal{B},\mathcal{V}}$.

- If the $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$ game finds BB invalid, $\mathcal{B}$ is then asked to guess a bit. It sends `no tally` to $\mathcal{E}$, who will output a bit. $\mathcal{B}$ applies the distinguisher D to $\mathcal{E}$'s output, and returns the result to $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$.

- Otherwise, $\mathcal{B}$ is then provided by $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},\beta}_{\mathcal{B},\mathcal{V}}$ with an oracle $\mathcal{O}\mathsf{verify}_{\mathsf{BB}}$ to make voters verify. $\mathcal{B}$ obtains from $\mathcal{A}$ the order in which verifications should be performed, and calls $\mathcal{O}\mathsf{verify}_{\mathsf{BB}}$ on each voter in $\mathsf{H}_{\mathsf{check}}$ following this order. If some verifications have failed, here too, $\mathcal{B}$ is asked for its guess for the bit β, which it obtains by sending `no tally` to $\mathcal{E}$, and applying D to $\mathcal{E}$'s output.

- Otherwise the game goes on to the tallying phase, and $\mathcal{B}$ is provided with the tallying oracle. $\mathcal{B}$ calls the tallying oracle, and obtains a result (r, Π). $\mathcal{B}$ sends (r, Π) to $\mathcal{A}$, asking if the result should be published. Depending on $\mathcal{A}$'s response, $\mathcal{B}$ either sends r or `no tally` to $\mathcal{E}$, and waits for its output.

- During the execution, $\mathcal{B}$ forwards between $\mathcal{E}$ and $\mathcal{A}$ any messages they wish to exchange.

- When $\mathcal{E}$ stops, $\mathcal{B}$ runs the distinguisher D on the output of $\mathcal{E}$, and answers to the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},\beta}_{\mathcal{B},\mathcal{V}}$ the guess D returns.

It is clear from the definitions of $\mathsf{realexec}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},\beta}_{\mathcal{B},\mathcal{V}}$ that, if $\beta = 0$, $\mathcal{B}$ provides the $\mathcal{E}$ and $\mathcal{A}$ it runs internally with the same view they would have in the corresponding execution of $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$. Similarly, if $\beta = 1$, the views of $\mathcal{E}$ and $\mathcal{A}$ as run by $\mathcal{B}$ are the same as their view in Game 1. Hence, the output of $\mathcal{E}$ (run by $\mathcal{B}$) is the output of Game 0 or Game 1, depending on β. Thus, whenever D correctly distinguishes Game 0 and Game 1, $\mathcal{B}$ correctly guesses β. Therefore, the advantage of a distinguisher between the outputs of Game 0 and Game 1 is at most the advantage of an adversary in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover},\beta}_{\mathcal{V}}$, and is thus negligible.

Game 2. establishes an invariant on BB'. It is as Game 1, but has $\mathcal{H}$ store, in addition to BB_0 and BB_1, the list of votes submitted in clear in BB_2. Once $\mathcal{H}$ has determined π, it additionally computes the associated tampering function $f = \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$, and applies it to BB_2 to obtain a list BB'_2. It then checks that BB'_2 correctly contains the votes in the ballots of BB'. Only in that case $\mathcal{H}$ lets the execution go through.

More precisely, Game 2 is identical to Game 1, except:

- on $\mathtt{vote}(id, v)$ from $\mathcal{E}$ for some $id \in \mathsf{H}$:
  in addition to computing ballots for v, v^* and sending them to the other parties, $\mathcal{H}$ also stores (id, v) into a list BB_2.

- on obtaining the board BB from $\mathcal{A}$:
  $\mathcal{H}$ computes $f = \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$ (recall that $\pi = \textsc{Recover}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$). $\mathcal{H}$ then lets $\mathsf{BB}'_2 = \overline{f}(\mathsf{BB}_2)$. Before sending BB' to $\mathcal{T}$ for tallying, $\mathcal{H}$ checks that $\mathsf{BB}'_2 = \overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$, and if not, halts the execution.

Game 1 and 2 can only be distinguished when the added check $\mathsf{BB}'_2 = \overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$ fails. We show by the first point of strong consistency that this has only negligible probability.

We actually show the stronger statement that $\mathsf{BB}''_2 = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$ except with negligible probability, where $\mathsf{BB}''_2 = \overline{f'}(\mathsf{BB}_2)$, and f' is such that for all i, $f'(i) = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \pi(i))$ if $\pi(i)$ is a ballot, and $f'(i) = \pi(i)$ otherwise. Note that by construction, $f = \lambda i.L'[i]$, where $L' = [f'(i), i \in \mathrm{dom}(f) | f'(i) \neq \bot]$ is the list obtained by removing all $\bot$ elements from f'. Hence, BB'_2 is obtained from BB''_2 by removing all $\bot$ elements. Since, by definition, $\overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$ is also obtained from $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$ by removing all $\bot$ elements, $\mathsf{BB}''_2 = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$ indeed implies $\mathsf{BB}'_2 = \overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$.

We construct an adversary $\mathcal{B}$ that breaks strong consistency with the same probability as the probability that $\mathsf{BB}''_2 \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$ in Game 2.

$\mathcal{B}$ runs Game 2 internally, simulating $\mathcal{E}$, $\mathcal{A}$, $\mathcal{H}$, ... by itself. If $\mathsf{BB}''_2 \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$, $\mathcal{B}$ retrieves a ballot $(p, b) \in \mathsf{BB}'$ that makes this test fail. That is, $\mathsf{BB}'[i] = (p, b)$ for some i, and $\mathsf{BB}''_2[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$. $\mathcal{B}$ then simply returns this ballot.

Intuitively, this ballot cannot be a dishonest one, as BB''_2 contains their extraction by construction; it is thus a honestly constructed ballot, whose extraction is not the vote used to construct it. This breaks strong consistency.

Formally, π is $\textsc{Recover}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$. By definition of $\textsc{Recover}$, either $\pi(i) = j$ for some j, or $\pi(i)$ is a ballot. The second case is actually impossible: indeed, in that case, $\mathsf{BB}'[i] = \pi(i)$ by definition of $\overline{\pi}$, and $f'(i) = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$ by definition of f'. Hence $\mathsf{BB}''_2[i] = f'(i) = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$, This contradicts the assumption that $\mathsf{BB}''_2[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$.

Hence, $\pi(i) = j$ for some j. By construction, $\mathsf{BB}' = \overline{\pi}(\mathsf{BB}_0)$. Thus by definition of $\overline{\pi}$, $\mathsf{BB}_0[j] = (id, \mathsf{BB}'[i]) = (id, (p, b))$ for some id. By construction of f', we also have $f'(i) = j$. Then, since $\mathsf{BB}''_2 = \overline{f'}(\mathsf{BB}_2)$, we have $\mathsf{BB}''_2[i] = \mathsf{BB}_2[j] = (id', v)$ for some id', v. By construction of BB_0 and BB_2, $\mathsf{BB}_2[j] = (id', v)$ was added to BB_2 by $\mathcal{H}$ at the same time $(id, (p, b))$ was added to BB_0, which means that $id = id'$, and (p, b) was generated by voter id calling $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$. Since $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) \neq \mathsf{BB}''_2[j]$, we have $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) \neq (id, v)$, which breaks strong consistency.

Therefore, the advantage of a distinguisher between the outputs of Game 1 and Game 2 is at most the advantage of an adversary against strong consistency, and is thus negligible.

Game 3. then makes use of this invariant: it is as Game 2 except $\mathcal{H}$ directly computes the result as $r = \rho(\mathsf{BB}'_2)$, instead of calling $\mathcal{T}$. Since this was the only action performed by $\mathcal{T}$ in Game 2, $\mathcal{T}$ is actually absent from Game 3.

By the previous invariant, and the second point of the strong consistency property, this yields the correct result except with negligible probability, and the two games are indistinguishable.

Indeed, the invariant introduced in Game 2 ensures that the tally is only computed if $\mathsf{BB}'_2 = \overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$. The outputs of Game 2 and Game 3 then only differ if $r' \neq r$, where $(r', \Pi') = \mathsf{Tally}(\mathsf{BB}', \mathsf{sk})$, that is, if $r' \neq \rho(\overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}'))$.

We can construct an adversary $\mathcal{B}$ that breaks strong consistency with the same probability as this occurring. $\mathcal{B}$ receives the key pk and the credentials U. It internally runs the setup and voting phases of Game 3 using pk and U, running $\mathcal{E}$, $\mathcal{A}$, $\mathcal{H}$ by itself. $\mathcal{B}$ retrieves the board BB' in this execution, and returns it: the condition $r' \neq \rho(\overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}'))$ is exactly the condition under which $\mathcal{B}$ wins the strong consistency game.

Hence, the advantage of an adversary distinguishing the outputs of Game 2 and Game 3 is at most the advantage of an adversary against strong consistency, and is thus negligible.

Game 4. removes the invariant introduced in Game 2. It is identical to Game 3, except that $\mathcal{H}$ does not check that $\mathsf{BB}'_2 = \overline{\mathsf{extract}}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})$ before computing the result.

The outputs of Game 3 and Game 4 are indistinguishable thanks to the first point of the strong consistency assumption, which follows from exactly the same reasoning as the hop between Game 1 and Game 2.

Game 5. entirely removes the ballots containing the real votes, as these are now no longer used to compute the result. $\mathcal{H}$ now only keeps track of BB_1 and BB_2, but not BB_0:

- on $\mathtt{vote}(id, v)$ from $\mathcal{E}$ for $id \in \mathsf{H}$:
  $\overline{\mathcal{H} \text{ only computes } (p', b', state'_{id})} = \mathsf{Vote}(\mathsf{pk}, id, c_{id}, v^*)$ (and sends $(id, (p', b'))$ to $\mathcal{A}$), and not $\mathsf{Vote}(\mathsf{pk}, id, c_{id}, v)$. As in Game 4, $\mathcal{H}$ still updates the lists BB_1 (with $(id, (p', b'))$) and BB_2 (with (id, v)), but no longer keeps BB_0.

As these ballots were unused, removing them has no bearing on the execution, and Game 5 is thus indistinguishable from Game 4. To sum up, the execution of Game 5 is as follows.

1. $\mathcal{E}$ sends $\mathtt{setup}$ to $\mathcal{H}$ to start the setup phase. $\mathcal{H}$ performs the setup: it generates $(\mathsf{pk}, \mathsf{sk})$ using Setup, and sends pk to the other entities $\mathcal{A}$, $\mathcal{R}$. For each $id \in \mathcal{I}$, $\mathcal{H}$ then sends $\mathtt{register}(id)$ to $\mathcal{R}$.
2. On $\mathtt{register}(id)$ from $\mathcal{H}$, $\mathcal{R}$ lets $c_{id} = \mathsf{Register}(id)$, and sends c_{id} to $\mathcal{H}$, who records it.

3. Once all voters have been registered, $\mathcal{H}$ sends the list of $(id, \mathsf{Pub}(c_{id}))$ for all id to $\mathcal{A}$. $\mathcal{H}$ also sends all dishonest credentials $[(id, c_{id}) | id \in \mathsf{D}]$ to $\mathcal{A}$, and gives control back to $\mathcal{E}$.

4. $\mathcal{E}$ then sends `voting` to $\mathcal{H}$ (and gets control back), starting the voting phase. It may then send any number of messages $\texttt{vote}(id, v)$ to $\mathcal{H}$ for any v and $id \in \mathsf{H}$.

5. On $\texttt{vote}(id, v)$ from $\mathcal{E}$, $\mathcal{H}$ computes a fake ballot $(p', b', state'_{id}) = \mathsf{Vote}(\mathsf{pk}, id, c_{id}, v^*)$, stores $(id, (p', b'))$ in BB_1 and (id, v) in BB_2, and sends $(id, (p', b'))$ to $\mathcal{A}$.

6. At some point, $\mathcal{E}$ sends `voting done` to $\mathcal{H}$. $\mathcal{H}$ then asks $\mathcal{A}$ for a board. $\mathcal{A}$ computes some bulletin board BB, and sends it to $\mathcal{H}$.

7. $\mathcal{H}$ then checks whether $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$. If so, it continues the execution, otherwise it sends `no tally` to $\mathcal{E}$.

8. $\mathcal{H}$ then obtains from $\mathcal{A}$ the order in which to verify. For each $id \in \mathsf{H}_{\mathsf{check}}$, in the specified order, $\mathcal{H}$ successively runs $\mathsf{Verify}(id, state'_{id}, \mathsf{BB})$.

9. Once all verifications have been performed, if any of them failed, $\mathcal{H}$ sends `no tally` to $\mathcal{E}$. Otherwise, $\mathcal{H}$ computes $\pi = \textsc{Recover}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$, $f = \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$, and $\mathsf{BB}'_2 = \overline{f}(\mathsf{BB}_2)$. $\mathcal{H}$ then computes $r = \rho(\mathsf{BB}'_2)$ and $\Pi' = \mathsf{SimProof}(r, \mathsf{BB})$.

10. $\mathcal{H}$ then sends $(r, \Pi'), \texttt{tally?}$ to $\mathcal{A}$.

11. $\mathcal{A}$ may then answer either `res-ok` or `res-block` to $\mathcal{H}$.

12. Following $\mathcal{A}$'s request, $\mathcal{H}$ then either sends r or `no tally` to $\mathcal{E}$.

13. Finally, $\mathcal{E}$ outputs a bit β. This bit is the output of Game 5.

Ideal adversary. We can now finally define the simulator $\mathcal{S}$, that runs $\mathcal{A}$ internally as a black-box, taking care by itself of most of the actions that were performed by $\mathcal{H}$ and $\mathcal{R}$ in Game 5, *i.e.* the cryptographic setup of keys and credentials, generation of fake ballots, computing of π and f. The only thing $\mathcal{S}$ cannot do on its own is store BB'_2 and apply f to it, as it does not know the real votes. Instead $\mathcal{S}$ sends the tampering function f to $\mathcal{F}_{\mathsf{v}}^P(\rho)$, who accepts it as Recover is allowed by P. $\mathcal{F}_{\mathsf{v}}^P(\rho)$ will then apply f to the list of honest votes, and compute the result, that $\mathcal{S}$ will show $\mathcal{A}$ together with a simulated proof.

Formally $\mathcal{S}$ answers commands as follows:

- On `setup` from $\mathcal{F}_{\mathsf{v}}^P(\rho)$:
  $\mathcal{S}$ runs Setup to obtain $(\mathsf{pk}, \mathsf{sk})$, and $\mathsf{Register}(id)$ for each $id \in \mathcal{I}$. It stores the generated secret and public credentials (computed with Pub) in tables U, PU. It also stores the credentials of dishonest voters into CU. $\mathcal{S}$ internally runs $\mathcal{A}$ in this simulated election: it transmits pk, CU and PU to $\mathcal{A}$.

- On `voting` from $\mathcal{F}_{\mathsf{v}}^P(\rho)$:
  $\mathcal{S}$ gives control back to $\mathcal{E}$.

- On $\texttt{ack}(id)$ from $\mathcal{F}_{\mathsf{v}}^P(\rho)$ during the voting phase:
  $\mathcal{S}$ generates a fake ballot for id: $(p', b', state'_{id}) = \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v^*)$. $\mathcal{S}$ records $state'_{id}$, transmits $(id, (p', b'))$ to $\mathcal{A}$, and stores $(id, (p', b'))$ in BB_1.

- On `voting done` from $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$:
  $\mathcal{S}$ gives control back to $\mathcal{E}$.

- On `modif?` from $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$:
  $\mathcal{S}$ asks $\mathcal{A}$ for a board, and obtains BB. $\mathcal{S}$ then checks that the board is valid, *i.e.* $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$, and, if so, asks $\mathcal{A}$ for the order in which verifications should be performed. $\mathcal{S}$ then runs $\mathsf{Verify}(id, state'_{id}, \mathsf{BB})$ for each $id \in \mathsf{H}_{\mathsf{check}}$, in the order chosen by $\mathcal{A}$. If the validity check on the board and all voter verifications succeed, $\mathcal{S}$ will then compute the tally (next point).

  Otherwise, $\mathcal{S}$ must prevent the publication of the result. $\mathcal{S}$ sends $\texttt{modif}(f_\emptyset)$ to $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$, where $f_\emptyset$ is the empty tampering function $f_\emptyset : \emptyset \to \emptyset$. Either the functionality accepts this tampering function, and computes a result, or it refuses it. In either case, $\mathcal{S}$ receives $\texttt{result}(r)$ from $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ for some r (which may be `no tally`). $\mathcal{S}$ then answers `res-block` to $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$. $\mathcal{H}'$, and then the environment $\mathcal{E}$, will then receive `no tally`.

- If all verifications succeeded, $\mathcal{S}$ must compute the tally, to simulate it for $\mathcal{A}$. $\mathcal{S}$ computes $\pi = \textsc{Recover}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$, and $f = \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$. $\mathcal{S}$ then sends $\texttt{modif}(f)$ to $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$. Let L_{id} be the list of *id*s contained in BB_1. By construction of BB_1, L_{id} is the list of the identities of all honest voters for whom `vote` queries were submitted, in the same order. Note that, since $\mathcal{E}$ is well-behaved, if this point is reached, $\mathcal{E}$ has requested each voter in $\mathsf{H}_{\mathsf{check}}$ to vote at least once. Hence BB_1, and L_{id} as well, contain at least one entry for each $id \in \mathsf{H}_{\mathsf{check}}$.

  In addition, by definition of the ideal execution, L_{id} is also the list of all *id*s occurring in the list L of votes recorded by $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$. By assumption, Recover is compatible with P. Hence, by definition, since BB is valid, π is compatible with P *w.r.t.* sk, U, and L_{id}, except with negligible probability. Indeed, an adversary could otherwise win the $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{comp},P,\textsc{Recover}}$ game by running internally $\mathcal{E}$, $\mathcal{A}$, $\mathcal{S}$, and submitting to the game the board BB. Thus, $P(L, f) = \top$, except with negligible probability, which means $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ accepts the modifications submitted by $\mathcal{S}$.

- On $\texttt{result}(r)$ from $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$, for some result $r \neq$ `no tally`:
  $\mathcal{S}$ computes $\Pi' = \mathsf{SimProof}(r, \mathsf{BB})$, and sends $(r, \Pi'), \texttt{tally?}$ to $\mathcal{A}$. $\mathcal{A}$ either answers `res-ok` or `res-block`. $\mathcal{S}$ forwards this message to $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$.

- In addition, during the whole execution, $\mathcal{S}$ forwards between $\mathcal{E}$ and $\mathcal{A}$ any message they wish to exchange.

It is clear that in the ideal execution with this simulator $\mathcal{S}$, both $\mathcal{E}$ and $\mathcal{A}$ have the same view as in Game 5, provided π is compatible with P *w.r.t.* sk, U. As explained above, this condition holds except with negligible probability. Hence, the outputs of Game 5 and $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P}(\rho))$ are indistinguishable, which concludes the proof. Therefore, the outputs of $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P}(\rho))$ and $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$ are indistinguishable, which concludes the proof. □

We can also state a variant of this theorem, for the case where revote is not allowed.

Theorem 12. *Let P be a predicate, and* Recover *a recovery algorithm compatible with P. Let $\mathcal{V}$ be a strongly consistent voting scheme for counting function ρ.*

If $\mathcal{V}$ satisfies mb-BPRIV w.r.t. Recover *with the restriction that the adversary may only call oracle $\mathcal{O}\mathsf{voteLR}$ at most once for each id, then $\mathcal{V}$ securely implements $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$, when considering only environments $\mathcal{E}$ that make each voter vote at most once.*

Proof. The proof of this theorem is exactly the same as for the previous one, except that in the hop between Games 0 and 1 we need to make sure that the adversary $\mathcal{B}$ that plays game $\mathsf{Exp}^{\text{mb-BPRIV},\text{RECOVER},\beta}_{\mathcal{B},\mathcal{V}}$ makes at most one call to oracle $\mathcal{O}\mathsf{voteLR}$ for each id. By definition of $\mathcal{B}$, a call to $\mathcal{O}\mathsf{voteLR}(id, v, v^*)$ is made whenever $\mathcal{E}$ sends a query $\mathtt{vote}(id, v)$, and only then. By assumption on $\mathcal{E}$, $\mathcal{E}$ does not make voters revote, and therefore indeed makes at most one such query for each id. □

The warm-up theorem stated earlier (Theorem 10) is a particular case of our main theorem (Theorem 11), applied to the predicates $P^{\emptyset}$, $P^{\mathsf{del,reorder}}$, $P^{\mathsf{del,reorder,change}}$ and the associated recovery algorithms $\text{RECOVER}^{\emptyset}$, $\text{RECOVER}^{\mathsf{del,reorder}}$, $\text{RECOVER}^{\mathsf{del,reorder,change}}$.

It directly follows from Theorem 11, provided we show that each of these recovery algorithms is compatible with the corresponding predicate.

Lemma 23 (Compatibility of the example recovery algorithms)**.** *Consider a strongly consistent voting scheme $\mathcal{V}$ for counting function ρ which does not produce duplicate ballots. For each* $\mathsf{power} \in \{\emptyset, (\mathsf{del}, \mathsf{reorder}), (\mathsf{del}, \mathsf{reorder}, \mathsf{change})\}$, $\text{RECOVER}^{\mathsf{power}}$ *is compatible with* P^{power}.

Proof. We prove this property, in all three cases, by considering an adversary $\mathcal{A}$, that will play the game $\mathsf{Exp}^{\mathsf{comp},P^{\mathsf{power}},\text{RECOVER}^{\mathsf{power}}}_{\mathcal{A},\mathcal{V}}$.

Following this game (the three games follow the same structure), let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, $\mathsf{U} \leftarrow \mathsf{Register}(1^\lambda, \mathcal{I})$, $\mathsf{CU} \leftarrow [\mathsf{U}[id] | id \in \mathsf{D}]$, and $\mathsf{PU} \leftarrow [\mathsf{Pub}(\mathsf{U}[id]) | id \in \mathcal{I}]$. $\mathcal{A}$ has access to the $\mathcal{O}\mathsf{vote}$ oracle, to generate honest ballots, that get stored in a board BB_1. For each $(id, (p, b)) \in \mathsf{BB}_1$, by construction, (p, b) was constructed by calling $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$ for some v. Let BB be the board returned by $\mathcal{A}$ in the game. Note that, by the assumption that Vote creates no duplicate ballots, BB_1 contains no duplicate ballots either, except with negligible probability. If BB is not valid, or BB_1 does not contain at least one entry for each $id \in \mathsf{H}_{\mathsf{check}}$, $\mathcal{A}$ loses the game.

Otherwise, let $\pi_1 = \text{RECOVER}^{\emptyset}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$, $\pi_2 = \text{RECOVER}^{\mathsf{del,reorder}}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$, and lastly $\pi_3 = \text{RECOVER}^{\mathsf{del,reorder,change}}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$. Let us show that π_1, π_2, π_3 are compatible respectively with $P^{\emptyset}$, $P^{\mathsf{del,reorder}}$, $P^{\mathsf{del,reorder,change}}$, *w.r.t.* $\mathsf{sk}, \mathsf{U}, \mathsf{L}_{id}$, except with negligible probability. Assume BB_1 contains no duplicate ballots, which, as explained, holds with overwhelming probability.

1. We first show that for any L such that $[id | (id, v) \in L] = \mathsf{L}_{id}$, $P^{\emptyset}(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)) = \top$.

   Consider the list LL constructed by the following process:

   $$\begin{array}{l} LL \leftarrow [1, \ldots, |\mathsf{BB}_1|]; \\ \textbf{for } (p, b) \in \mathsf{BB} \textbf{ do} \\ \quad \textbf{if } \mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H} \textbf{ then} \\ \quad\quad LL \leftarrow LL \parallel \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) \end{array}$$

   By definition, $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)$ is $\lambda i.LL'[i]$, where LL' is the list obtained obtained by removing all $\perp$ elements from LL. We have to show that $P^{\emptyset}(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)) = \top$. That is, that

   - No honest votes are modified by $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)$:

     $$\forall i.\ \forall (id, v).\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)[i] = (id, v) \implies id \in \mathsf{D}.$$

     Let i, id, v be such that $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)[i] = (id, v)$. Hence $LL[j] = (id, v)$ for some j. Thus, by construction of LL, $id \in \mathsf{D}$.

- $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)$ keeps the votes of all honest voters in the same order:

$$[\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)(j) \mid \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_1)(j) \in \mathbb{N}] = [1, \ldots, |L|]$$

that is, since the non-$\perp$ elements in LL' and LL are in the same order,

$$[LL[j] \mid LL[j] \in \mathbb{N}] = [1, \ldots, |L|]$$

which clearly holds by construction of LL.

2. We now show that for any L such that $[id|(id, v) \in L] = \mathsf{L}_{id}$, $P^{\mathsf{del},\mathsf{reorder}}(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)) = \top$. Consider the lists LL, LL', LL'' constructed by the following process:

$LL \leftarrow []$;
for $(p, b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $LL \leftarrow LL \parallel j$
    (by assumption on BB_1, this j is unique)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ **then**
    $LL \leftarrow LL \parallel \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$
$LL' \leftarrow [i|\mathsf{BB}_1[i] = (id, (p, b)) \ \wedge \ id \in \mathsf{H}_{\mathsf{check}} \ \wedge \ (p, b) \notin \mathsf{BB}]$
$LL'' \leftarrow LL \parallel LL'$

By definition, $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)$ is $\lambda i.LL'''[i]$, where LL''' is the list obtained obtained by removing all $\perp$ elements from LL''. We have to show that $P^{\mathsf{del},\mathsf{reorder}}(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)) = \top$. That is, that

- No honest votes are modified by $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)$:

$$\forall i.\ \forall (id, v).\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)[i] = (id, v) \implies id \in \mathsf{D}.$$

Let i, id, v be such that $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)[i] = (id, v)$. Hence $LL[j] = (id, v)$ for some j. Thus, by construction of LL, $id \in \mathsf{D}$.

- $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)$ keeps the votes of all honest voters who check:

$$\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall v.\ L[i] = (id, v) \implies \exists j.\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)(j) = i,$$

that is,

$$\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall (p, b).\ \mathsf{BB}_1[i] = (id, (p, b)) \implies \exists j.\ LL''[j] = i.$$

Let $i, p, b, id \in \mathsf{H}_{\mathsf{check}}$ such that $\mathsf{BB}_1[i] = (id, (p, b))$. Either $(p, b) \notin \mathsf{BB}$, and then by definition of LL', $i \in LL'$. Or $(p, b) \in \mathsf{BB}$, and by construction $i \in LL$. In any case, $i \in LL''$.

3. We finally show that for any L such that $[id|(id, v) \in L] = \mathsf{L}_{id}$, $P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_3)) = \top$.

Consider the lists LL, LL', LL'' constructed by the following process:

$LL \leftarrow []$;
for $(p, b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $LL \leftarrow LL \parallel j$
    (by assumption on BB_1, this j is unique)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}_{\mathsf{check}}$ **then**
  $LL \leftarrow LL \parallel \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$
$LL' \leftarrow [i | \mathsf{BB}_1[i] = (id, (p, b)) \ \wedge \ id \in \mathsf{H}_{\mathsf{check}} \ \wedge \ (p, b) \notin \mathsf{BB}]$
$LL'' \leftarrow LL \parallel LL'$

By definition, $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_3)$ is $\lambda i.LL'''[i]$, where LL''' is the list obtained obtained by removing all $\perp$ elements from LL''. We have to show that $P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_3)) = \top$. That is, that

- No votes from voters who verify are modified by $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_3)$:

$$\forall i.\ \forall (id, v).\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_3)[i] = (id, v) \implies id \in \mathsf{D}.$$

  Let i, id, v be such that $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_2)[i] = (id, v)$. Hence $LL[j] = (id, v)$ for some j. Thus, by construction of LL, $id \in \mathsf{D} \cup \mathsf{H}_{\overline{\mathsf{check}}}$.

- $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_3)$ keeps the votes of all honest voters who check:

$$\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall v.\ L[i] = (id, v) \implies \exists j.\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi_3)(j) = i,$$

  that is,

$$\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall (p, b).\ \mathsf{BB}_1[i] = (id, (p, b)) \implies \exists j.\ LL''[j] = i.$$

  Let $i, p, b, id \in \mathsf{H}_{\mathsf{check}}$ such that $\mathsf{BB}_1[i] = (id, (p, b))$. Either $(p, b) \notin \mathsf{BB}$, and then by definition of LL', $i \in LL'$. Or $(p, b) \in \mathsf{BB}$, and by construction $i \in LL$. In any case, $i \in LL''$.

□

6.6 Application to voting schemes

We use the general framework developed in previous sections to study the resilience of three protocols of the literature, namely Helios [64], Belenios [57], and Civitas [54], in the presence of malicious boards. For each of them, we identify which ideal functionalities they achieve. Interestingly, the guarantees differ depending on the revote policy in place and the counting function ρ.

We consider security with respect to the three functionalities introduced in Section 6.4.2 plus a new functionality we introduce here. $\mathcal{F}_{\mathsf{v}}^{\emptyset}$ is the very ideal functionality where honest votes are registered and processed exactly as they are received, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$ (which is the functionality associated with the predicate P^{del} defined in Figure 6.10) lets the adversary remove some honest votes (from voters that do not check), but requires that the votes of voters who check are kept and not reordered, while $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$ further lets the adversary reorder the votes. Finally, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$ even lets the adversary modify honest votes, from voters who do not verify.

$P^{\mathsf{del}}(L, f) = \top$ iff:

- f keeps the votes of all voters who check, in the same order for each voter:
  $\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i \in [\![1, |L|]\!] | \exists v.\ L[i] = (id, v)] =$
  $[f(j), j = 1 \ldots |\mathrm{dom}(f)| | \exists v.\ L[f(j)] = (id, v)]$
- and no honest votes are modified by f:
  $\forall i, id, v.\ f(i) = (id, v) \implies id \in \mathsf{D}$.

Figure 6.10 – Predicate P^{del}, associated with functionality $\mathcal{F}_{\mathrm{v}}^{\mathsf{del}}(\rho)$.

6.6.1 Overview of the protocols

Helios. Helios is the simple voting protocol we already presented and studied in earlier chapters (*e.g.* Chapter 3). It aims at guaranteeing privacy and verifiability in a low-coercion environment. It has been used in several elections, *e.g.* to elect the president of the university of Louvain-la-Neuve [11], or for student elections at Princeton [91].

In Helios, voters cast a vote by computing a ballot $\mathsf{Vote}(id, \mathsf{pk}, c, v) = (s, id, (\mathsf{enc}(v, \mathsf{pk}), \pi))$, where the state s records the ciphertext $(\mathsf{enc}(v, \mathsf{pk}), \pi)$ and id is the identity of the voter (or possibly a pseudonym). If the voter id votes again then the state is updated with the new ciphertext. The ciphertext is simply formed of an ElGamal encryption $\mathsf{enc}(v, \mathsf{pk})$ of v under the public key of the election pk, together with π, a zero-knowledge proof that guarantees that v is a valid vote. Helios does not use credentials, hence c is not used. The check $\mathsf{Verify}(id, s, \mathsf{BB})$ done by a voter id consists in verifying that the ciphertext recorded in s appears on BB.

The $\mathsf{ValidBoard}$ algorithm checks all the zero-knowledge proofs in each ballot in BB. In addition Helios features a weeding process, *i.e.* the operation of rejecting duplicate ciphertexts, as already described in Chapter 3. We leave this process to the adversary, and encode this by simply having the $\mathsf{ValidBoard}$ algorithm reject any board submitted by the adversary that contains duplicate ciphertexts. In cases where we want to consider an election where revote is not allowed, the $\mathsf{ValidBoard}$ algorithm will also reject any board with several ballots for the same voter identity.

Finally, the tally can either be mixnet-based or homomorphic. In the mixnet case, it performs a random permutation of the ballots, decrypts all of them and returns the multiset of the votes they contain. In the homomorphic mode, the tally homomorphically computes the sum of the ciphertexts in the ballots in BB, decrypts the resulting ciphertext and returns the result. Moreover, the tally returns a proof of correct decryption.

Belenios. Belenios [57] enhances Helios with credentials, so that a compromised voting server cannot add votes. It has been launched in 2016 and used in more than 200 elections [59]. At registration, each voter id receives a signing key k_{id}, with an associated verification key pk_{id}. The voting procedure $\mathsf{Vote}(id, \mathsf{pk}, (k_{id}, pk_{id}), v)$ produces the state and ballot $(s, pk_{id}, (\mathsf{signElGamal}(v, \mathsf{pk}, k_{id}), \pi))$, where $\mathsf{signElGamal}(v, \mathsf{pk}, k_{id})$ denotes the ElGamal encryption of v, signed with k_{id}. The other algorithms are modified as expected.

To ease the verification step made by voters, in Helios and Belenios, only the last ballot for each voter is presented in the bulletin board. This can be modelled by a $\mathsf{Verify}(id, s, \mathsf{BB})$ algorithm that checks that the last ballot recorded in s is the last ballot appearing in BB for voter id.

Formally, we model the scheme as the following algorithms.

- $\mathsf{Register}(id)$ generates a pair (k_{id}, pk_{id}) of a secret signing key k_{id} and the associated verification key pk_{id}.
- $\mathsf{Pub}(k_{id}, pk_{id}) = pk_{id}$ is simply the public verification key.
- $\mathsf{Vote}(id, \mathsf{pk}, (k_{id}, pk_{id}), v) = (s, pk_{id}, (\mathsf{signElGamal}(v, \mathsf{pk}, k_{id}), \pi))$. The state s records the ballot. The public credential/pseudonym is the public key pk_{id}. π is a zero-knowledge proof that v is a valid vote.
- $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk})$ checks for each (p, b) in BB the zero-knowledge proof in b, and ensures that p is a valid verification key, and that b is indeed signed with the associated signing key. When modelling a situation where revote is forbidden, it would finally check that BB does not contain several ballots (p, b), (p, b') that use the same public key, *i.e.* such that $p = p'$.
- Tally can either run the ballots through a mixnet, decrypt them, and publish the decrypted votes; or homomorphically compute the sum of all ballots, before decrypting and publishing the result.
- $\mathsf{Verify}(id, s, \mathsf{BB})$ checks whether the last ballot recorded in s, *i.e.* the last ballot generated by id, is the last ballot that appears next to the public key of id in BB, or, equivalently when the board is valid, the last ballot signed with id's key in BB.

Civitas. Civitas [54] has been designed to protect voters against coercion. Each voter privately receives a credential, and can produce a fake credential when she is under coercion. Ballots cast with invalid credentials are removed thanks to plaintext equality tests (PET), after some mixing phase, to avoid a coercer noticing that his ballots have been excluded.

- $\mathsf{Register}(id)$ generates a private credential c_{id}
- $\mathsf{Pub}(c) = \mathsf{enc}(c, \mathsf{pk})$ encrypts the credential c
- $\mathsf{Vote}(id, \mathsf{pk}, c, v) = (s, \mathsf{enc}(c, \mathsf{pk}), (\mathsf{enc}(v, \mathsf{pk}), \pi_1, \pi_2))$. The state s records the ballot. The public credential/pseudonym is $\mathsf{enc}(c, \mathsf{pk})$. π_1 is a zero-knowledge proof that v is a valid vote, and π_2 is a zero-knowledge proof that the agent generating the ballot knows both c and v. In Civitas, the voting server can no longer select the "last" ballot for each voter since ballots cannot be properly linked to an identity. So when revote is allowed, a voter who revotes should additionally link her new ballot to the previous one (*e.g.* adding a copy of the previous ballot, or its hash), and prove (with zero-knowledge proofs) that she knows the credential and the choices used in both ballots.
- $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk})$ checks for each (p, b) in BB the zero-knowledge proofs in b, and ensures that p is a valid public credential, using $\mathsf{Pub}(\mathsf{U})$ and a PET. In case we model a situation where revote is not allowed, it additionally checks, also with a PET, that BB does not already contain several ballots (p, b), (p', b') where p and p' encrypt the same credential.
- Tally only keeps the parts of the ballots containing the encrypted votes. These are then run through a mixnet, decrypted, and published.
- $\mathsf{Verify}(id, s, \mathsf{BB})$ checks whether the ballot recorded in s, *i.e.* the last ballot generated by id, appears on BB.

Voting scheme	$\mathcal{F}_{\mathsf{v}}^{\emptyset}$		$\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$	$\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$	$\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$
	General case	$\mathsf{H}_{\mathsf{check}} = \mathsf{H}$			
Helios - without revote	✗	✓ if ρ stable $\wedge$ *id*-blind	✗	✗	✓ if ρ *id*-blind
Helios - revote	✗	✗	✗	✗	✗
Belenios - without revote	✗	✓ if ρ stable	✓	✓	✓
Belenios - revote	✗	✓ if ρ stable $\wedge$ $\rho = last$	✓ if $\rho = last$	✓ if $\rho = last$	✓ if $\rho = last$
Civitas - without revote	✗	✓ if ρ stable	✓	✓	✓
Civitas - revote	✗	✓ if ρ stable	✓	✓	✓

Figure 6.11 – Case study.

6.6.2 Our findings

The results of our study are gathered in Figure 6.11. For each protocol, we distinguish the case where revote is allowed from the case where it is not. When revote is not allowed then we assume that (honest) voters do *not* revote. As we will discuss in this section, this is not equivalent, security-wise, to the case where voters may revote but only the first vote is counted.

(in)security of Helios. As mentioned already in introduction, Helios is subject to an attack [94] if the attacker controls the bulletin board (or simply the communication channel between the voter and the server). Indeed, an attacker may block and copy the first ballot b_A sent by Alice, say for candidate 0. The attacker can then pretend that the communication was lost, so that Alice starts over the procedure and sends again a ballot, b'_A, still for candidate 0 (there is no reason that she changes her mind). Then since ballots are not cryptographically authenticated in Helios, the attacker may submit b_A as his own ballot, introducing a bias in the result. This attack cannot be prevented, even if the auditors check for duplicates before the tally. Therefore Helios with revote does not satisfy any of the four functionalities.

Assume now that there is no revote. Since in Helios the identity of a voter is not strongly linked to the ciphertext containing her vote, an adversary is able to swap two voters' ciphertexts, *e.g.* replacing $[(A, b_A), (B, b_B)]$ with $[(A, b_B), (B, b_A)]$. This is fine as long as the counting function processes the votes independently of the actual identity of a voter, so that attributing Alice's vote to Bob and vice versa does not change the result. We call this property *id-blindness*. Most voting functions enjoy this property but not all of them. For example, for elections with weighted votes, which can be done homomorphically [11], each vote is associated to a weight depending on the status of the voter. For example, it could be the case that Alice's vote is counted 10 times while Bob's vote is counted only 3 times. For *id*-blind counting functions, then Helios satisfies $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$, the weakest functionality, since an attacker may reorder votes and remove and even modify the ballots of voters that do not check.

No revote vs $\rho = first$. Interestingly, the Helios example illustrates why it is not possible to properly emulate the "no revote" policy by letting voters revote and considering a function ρ where only the first ballot is counted for each voter. In fact, if voters may revote then the adversary has more power. In particular, Helios with revote is still subject to Roenne's attack, even if only the first ballot is counted.

No reordering in Civitas. Our findings highlight that Civitas is the only scheme that prevents an adversary from reordering the votes, thanks to the link made by voters between their ballots. Note that if the attacker controls the board, then he can always permute Alice and

Bob's ballots without anyone noticing. However, this does not influence the result for all the result functions ρ we know, namely *stable functions*, that is counting functions where reordering the votes does not influence the result (provided that the for each voter id, the order of all the votes of id is preserved). Formally, we say that ρ is stable if $\rho(L_1) = \rho(L_2)$ for any two lists L_1 and L_2 such that, for any voter id, if $(id, v_1), \ldots, (id, v_n)$ is the sequence of votes from voter id in L_1, then $(id, v_1), \ldots, (id, v_n)$ is also the sequence of votes from voter id in L_2 (in the same order). We believe that all existing result functions are stable, since intuitively whether Alice voted before Bob or after Bob should not influence the result.

So now the question is: which schemes can prevent an adversary from reordering the votes of a given voter? This would be a priori a real attack since it does change the result. Our results show that only Civitas protects again such a re-ordering, thanks to the chain between ballots cast by the same voter. Belenios provides weaker security guarantees since the adversary may reorder the votes. However, this does not affect the result as long as only the last ballot is counted, since Alice checks that her last ballot appears in the final board. To render Belenios suitable for *arbitrary* (stable) counting functions, we would need to require that each voter records her ballots in order, and checks that they appear *in the same order* in the final board. This would of course not be realistic. Alternatively, the most reasonable approach is probably to chain ballots thanks to an additional zero-knowledge proof, like in Civitas. Note however that this chain is only briefly sketched in [54] for Civitas, and no proper definition is provided.

"Perfect" functionality $\mathcal{F}_{\mathsf{v}}^{\emptyset}$. As one might expect, none of the three schemes satisfy in general the strongest ideal functionality, where the attacker cannot tamper at all with honest votes. This is due to the fact that an adversary can always drop the ballots of voters that do not check. This limitation applies to many other schemes as well. If we assume now that *all* honest voters actually vote and conduct all required checks, the three schemes (except Helios with revote) satisfy $\mathcal{F}_{\mathsf{v}}^{\emptyset}$. This requirement is however not realistic in practice.

Proofs. To establish security with respect to ideal functionalities in Figure 6.11 we leverage the framework we have developed in this chapter in two distinct ways. On the one hand, for each scheme in turn we prove game based security with respect to appropriately chosen recovery algorithms and then employ Theorem 11 (or Theorem 12 when revote is forbidden) to conclude simulation-based security. Depending on the protocol considered, we define additional recovery algorithm, that are associated to the same ideal functionality (*i.e.* compatible with the same predicate), but are better suited to the protocol studied. Examples of such algorithms, as well as the proofs for all examples in our case study, can be found in appendix B.

Interestingly, we also employ reasoning about ideal functionalities directly. Specifically, we show that $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}} \wedge \rho$ stable $\wedge\ \mathsf{H} = \mathsf{H}_{\mathsf{check}} \Rightarrow \mathcal{F}_{\mathsf{v}}^{\emptyset}$) and the desired results in the column $\mathsf{H} = \mathsf{H}_{\mathsf{check}}$ (under the $\mathcal{F}_{\mathsf{v}}^{\emptyset}$ heading) follow from those in column with heading $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$. The formal statement and the proof of this result are provided in appendix B.1.

6.6.3 Comparing privacy

We also take this opportunity to compare our mb-BPRIV notion with different existing notions of privacy, with and without an honest ballot box, on the three protocols listed above, as well as the Neuchâtel protocol [78]. We first briefly introduce that protocol, and then describe the different notions of privacy we compare.

The Neuchâtel protocol In that protocol, used in Switzerland, voters privately receive a code sheet, where each candidate is associated to a (short) code. To cast a vote, voters send their encrypted votes to the ballot box, similarly to Helios. The ballot box then provides a return code allowing the voter to check that the ballot has been received and that it encrypts their candidate, as intended. This offers a protection against a dishonest voting client (*e.g.* if the voter's computer is corrupted). No revote is allowed. Since the bulletin board is not published, voters cannot check that their ballots really belong to the final board (used for tally). Voters have to trust the entity generating the return code on this aspect. The model for voting systems we consider here cannot accurately describe voters having to receive a return code from some authority. Hence our mb-BPRIV definition cannot be applied as-is to that voting scheme, which is why we did not include it in the case study above.

Different notions of privacy We compare mb-BPRIV to privacy against a honest board (as formalised by Benaloh's definition or hb-BPRIV), as well as the naïve adaptation of Benaloh's definition to a dishonest board we described earlier, which we denote PrivDis-Naive. In that naïve definition, the adversary $\mathcal{A}$ may propose two possible affectations of votes for honest voters through an oracle $\mathcal{O}$voteLR, and then propose any board BB. Provided the votes submitted to the oracle yield the same tally on both sides, $\mathcal{A}$ obtains the tally of BB. He is then asked to guess which of the two affectations was used.

To our knowledge, the only other definition of privacy with a dishonest ballot box is the privacy notion introduced by Bernhard and Smyth [29], which we discuss below.

Bernhard and Smyth's definition As we discussed in introduction of this chapter, that definition is similar in spirit to Benaloh's definition (presented in Chapter 5). We recall it in Figure 6.12 (PrivacyBS). It takes the form of a game where the adversary $\mathcal{A}$ may propose two possible affectations of votes, *i.e.* has access to an oracle $\mathcal{O}^{bs}_{\mathsf{vote}}(id, v_0, v_1)$, that returns to $\mathcal{A}$ a ballot for either v_0 or v_1, depending on a secret bit β. The adversary is asked to produce a ballot box BB, and, as in Benaloh's definition – but contrary to mb-BPRIV – the tally is always computed on BB (*i.e.* on both sides). To avoid trivial attacks, following the same idea as Benaloh's definition, the tally is only performed provided that, looking at honest ballots that appear in BB, counting the corresponding left and right votes yields the same result.

The main interest of [29] is to highlight the fact that previous definitions implicitly assume an honest ballot box. That attempt at defining privacy *w.r.t.* a dishonest ballot box (PrivacyBS) however has several limitations. First, it strongly assumes that the ballots that appear in the ballot box are exactly the same than the cast ballots. This is not the case for example of the ThreeBallots protocol [93] where the ballot box only contains two shares (out of three) of the original ballot. It is not applicable either to a protocol like BeleniosRF [44] where ballots are re-randomised before their publication. Second, it requires ballots to be non-malleable [29]. This means that, as soon as a ballot includes a malleable part (for example the voter's id like in Helios, or a timestamp), privacy cannot be satisfied. This severely restricts the class of protocols that can be considered. Third, PrivacyBS does not account for a revote policy. As soon as revote is allowed (for example in Helios), then PrivacyBS is broken since some ballots may not be counted. Indeed, an attacker may call $\mathcal{O}^{bs}_{\mathsf{vote}}(id_1, 1, 0)$, followed by $\mathcal{O}^{bs}_{\mathsf{vote}}(id_1, 0, 1)$, obtaining ballots b_1, b'_1, and return the board $\mathsf{BB} = [b_1, b'_1]$. The equality condition on the number of ballots in BB produced by $\mathcal{O}^{bs}_{\mathsf{vote}}$ holds, since for $v = 0, 1$:

$$|\{b \in \mathsf{BB} | \exists v'.\ (b, v, v') \in \mathsf{L}\}| = |\{b \in \mathsf{BB} | \exists v'.\ (b, v', v) \in \mathsf{L}\}| = 1$$

$\mathsf{Exp}^{\mathsf{BS},\beta}_{\mathcal{A}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
$\mathsf{U},\mathsf{D} \leftarrow []$
$\mathcal{A}^{\mathcal{O}_{\mathsf{reg}},\mathcal{O}_{\mathsf{corr}}}(\mathsf{pk})$
$\mathsf{L} \leftarrow []$
$\mathcal{A}^{\mathcal{O}^{bs}_{\mathsf{vote}}}(\mathsf{pk})$
if $\forall v.$
$|\{b|b \in \mathsf{BB} \wedge \exists v'.\ (b,v,v') \in \mathsf{L}\}| =$
$|\{b|b \in \mathsf{BB} \wedge\ \exists v'.\ (b,v',v) \in \mathsf{L}\}|$
if $\rho(\mathcal{I}_0) = \rho(\mathcal{I}_1)$ **then**
  $r \leftarrow \mathsf{Tally}(\mathsf{BB},\mathsf{sk},\mathsf{U})$
  $\beta' \leftarrow \mathcal{A}(r)$
  return β'

$\mathcal{O}^{bs}_{\mathsf{vote}}(id,v_0,v_1)$

if $(id,*) \in \mathsf{U} \ \wedge\ id \notin \mathsf{D}$ **then**
  $b \leftarrow \mathsf{Vote}(\mathsf{pk},id,\mathsf{U}[id],v_\beta)$
  $\mathsf{L} \leftarrow \mathsf{L} \| (b,v_0,v_1)$
  return b

$\mathcal{O}_{\mathsf{corr}}(id)$

if $(id,*) \notin \mathsf{U} \ \vee\ id \in \mathsf{D}$ **then stop**
else
  $\mathsf{D} \leftarrow \mathsf{D} \| id$
  return $\mathsf{U}[id]$

$\mathcal{O}_{\mathsf{reg}}(id)$

if $(id,*) \in \mathsf{U}$ **then stop**
else
  $c \leftarrow \mathsf{Register}(1^\lambda,id)$
  $\mathsf{U}[id] \leftarrow c$
  return $\mathsf{Pub}(\mathsf{U}[id])$

Figure 6.12 – PrivacyBS [29]

Protocol	Honest board	PrivDis-Naive	PrivacyBS	mb-BPRIV		"Secure"
Helios – no revote	✓	✗	✗	✓	for $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$	✗
Helios – revote	✓	✗	✗	✗		✗
Belenios – no revote	✓	✗	✓	✓	for $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$	✓
Belenios – revote	✓	✗	✗	✓	for $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$	✓
Civitas – no revote	✓	✗	✓	✓	for $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$	✓
Civitas – revote	✓	✗	✗	✓	for $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$	✓
Neuchâtel	✓	✗	?	-		?

Figure 6.13 – Comparison of several privacy definitions
(✓: the protocol is private, ✗: there exists an attack on privacy, ?: unknown, -:definition does not apply)

where $\mathsf{L} = [(b_1,1,0),(b'_1,0,1)]$.

Hence the tally is computed. According to the revote policy, only b'_1 is counted, and the result is β, which lets the attacker win $\mathsf{Exp}^{\mathsf{BS}}$.

Comparison We summarise our findings in Figure 6.13. All of our four protocols satisfy privacy against an honest ballot box. We rely here on previous results of the literature. Regarding dishonest boards, we first consider the naïve extension of Benaloh's definition to a dishonest board (PrivDis-Naive), described above. As explained earlier, it is immediately violated for any protocol. We also include Bernhard and Smyth's PrivacyBS notion [29], which we also discussed above. Finally, we recall in the last column the strongest ideal functionality our mb-BPRIV notion allowed us to establish for each protocol in our case study.

We do not know whether the Neuchâtel protocol satisfies PrivacyBS – the protocol was never studied with that definition. Moreover that protocol cannot be accurately represented in our model – our definition does not apply to it. As discussed above, PrivacyBS is immediately violated as soon as revote is allowed.

In addition, we indicate in the rightmost column of Figure 6.13 whether, in our opinion, the protocol should be considered secure in the context of a malicious board. This is of course a subjective matter, but we think that Belenios and Civitas do satisfy privacy in that threat model, with or without revote. On the other hand, it is our opinion that Helios does not – when revote

is allowed, Helios is subject to Roenne's attack, for which no easy fix is known. Even without revote, an attacker against Helios is able to replace ballots from honest voters who do not verify with any ballot for arbitrary vote. Such a behaviour is prevented by both Belenios and Civitas, and we think that it should be considered an issue when the ballot box is not trusted. Regarding the Neuchâtel protocol, the situation is less clear: it seems to depend on whether we consider that the entity issuing the return codes is a part of ballot box – and thus untrusted – or not.

This comparison first shows that mb-BPRIV is much more precise than the other definitions, allowing to finely describe what security protocols guarantee. For instance, it does not broadly declare Helios (without revote) insecure, but rather allow us to show that, although it does not achieve the strongest guarantees, it still provide some security if we wish to consider that changing votes of voters who do not verify is not an attack. It also seems to indicate that our intuition for a secure scheme more or less corresponds to the ideal functionality $\mathcal{F}_{\mathsf{V}}^{\mathsf{del}}$, that prevents the adversary from removing, changing, or reordering votes of voters who do verify, and additionally from changing the votes of voters who do not. While this is one of the stronger functionalities we consider, it is still achievable, as shown by our case study, and seems like a reasonable goal in a context where the board is not trusted, and not all voters verify their vote.

6.7 mb-BPRIV implies verifiability

We established in Chapter 4 that (under reasonable assumptions) privacy implies individual verifiability, for Benaloh's privacy notion, that considers a honest ballot box. Interestingly, this implication still holds in the case of a malicious ballot box for our new mb-BPRIV notion, although not with the same proof.

Intuitively, the simulation-based notion of security means that a voting scheme implements the parametric ideal functionality $\mathcal{F}_{\mathsf{V}}^{P}(\rho)$ we defined earlier. While we designed this functionality with privacy in mind, it actually captures stronger guarantees than just privacy. It guarantees that the adversary cannot tamper with the votes in any other way than those authorised by the predicate P. With a well-chosen P, that forbids changing or removing the votes from voters who verify, this functionality thus also provides individual verifiability guarantees. In turn, by our main Theorem 11, mb-BPRIV also entails individual verifiability, provided the recovery algorithm considered is compatible with such a predicate.

6.7.1 Properties and assumptions

To prove this result formally, we first need to define individual verifiability against a malicious ballot box. We adapt the game-based definition from the previous chapter (Section 5.1.2), that modelled the case of an honest ballot box. Basically, as we did earlier for privacy, we modify the game so that the adversary can submit an arbitrary board, rather than only giving him access to the board through oracles. This yields the game $\mathsf{Exp}^{\mathsf{mb\text{-}verif}}$ displayed in Figure 6.14. As before, the after the setup phase, the adversary is given access to all public information, as well as the credentials of corrupt voters, and an oracle to choose the votes of honest voters. When a honest voter id votes for v, a ballot for v is honestly generated using id's credential, and returned to the adversary. The voter's identity and internal state are recorded in V, and in addition, if $id \in \mathsf{H}_{\mathsf{check}}$, then (id, v) is recorded in Voted. The adversary is asked to provide a board BB. If this board is invalid, or if not all voters supposed to verify have voted, the game halts. Otherwise, the adversary is given access to oracle $\mathcal{O}\mathsf{verify}_{\mathsf{BB}}(id)$, that triggers voter id's

$\mathsf{Exp}^{\mathsf{mb\text{-}verif}}_{\mathcal{A},\mathcal{V}}(\lambda)$

$\mathsf{V}, \mathsf{Voted}, \mathsf{Happy} \leftarrow []$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $c \leftarrow \mathsf{Register}(1^\lambda, id) \quad \mathsf{U}[id] \leftarrow c, \mathsf{PU}[id] \leftarrow \mathsf{Pub}(c)$
for all $id \in \mathsf{D}$ **do** $\mathsf{CU}[id] \leftarrow \mathsf{U}[id]$
$\mathsf{BB} \leftarrow \mathcal{A}^{\mathcal{O}\mathsf{vote}}(\mathsf{pk}, \mathsf{CU}, \mathsf{PU})$
if $\mathsf{H}_{\mathsf{check}} \not\subseteq \mathrm{dom}(\mathsf{V})$ **then return** $\bot$;
if $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \bot$ **then return** $\bot$;
$\mathcal{A}^{\mathcal{O}\mathsf{verify}_{\mathsf{BB}}}()$
if $\mathsf{H}_{\mathsf{check}} \not\subseteq \mathsf{Happy}$ **then return** $\bot$;
$r \leftarrow \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$
if $r \neq \bot \ \wedge\ \forall V_c.$
  $(\forall (id, v) \in V_c.\ id \notin \mathsf{H}_{\mathsf{check}}) \Rightarrow r \neq \rho(\mathsf{Voted} \parallel V_c)$
then return 1
else return 0.

$\mathcal{O}\mathsf{vote}(id, v)$ for $id \in \mathsf{H}$

$(p, b, state) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$
$\mathsf{V}[id] \leftarrow state$
if $id \in \mathsf{H}_{\mathsf{check}}$ **then** $\mathsf{Voted} \leftarrow \mathsf{Voted} \parallel (id, v)$
return (p, b).

$\mathcal{O}\mathsf{verify}_{\mathsf{BB}}(id)$ for $id \in \mathsf{H}_{\mathsf{check}}$

if $\mathsf{Verify}(id, \mathsf{V}[id], \mathsf{BB}) = \top$ **then**
  $\mathsf{Happy} \leftarrow \mathsf{Happy} \cup \{id\}$

Figure 6.14 – The individual verifiability game, against a dishonest ballot box.

verifications. If the verification is successful, *id* is added to the set Happy (initially empty). Once the adversary gives control back, provided all voters in $\mathsf{H}_{\mathsf{check}}$ are satisfied with BB (*i.e.* were added to Happy), BB is tallied. We finally check whether the result r contains at least all votes in Voted in that order (*i.e.* all votes from voters who verified). That is, we check whether there exists a list V_c of pairs (id, v) (with $id \notin \mathsf{H}_{\mathsf{check}}$) such that $r = \rho(\mathsf{Voted} \parallel V_c)$. If not, the game returns 1, and the adversary wins.

Note that in this definition, when writing $\rho(\mathsf{Voted} \parallel V_c)$, we grouped all votes from voters in $\mathsf{H}_{\mathsf{check}}$ in the beginning of the list (without changing their order), and all additional votes in the end. For this to make sense, we will assume that the counting function ρ is *stable*, in the sense defined in the previous section (*i.e.*, changing the order of votes does not change the result, as long as for any *id*, *id*'s votes remain in the same order).

As in the previous chapter, we will assume that given a result r and a sequence V of identities and votes, it is possible to efficiently decide whether r contains all votes in V. That is, we assume a polynomial-time algorithm D such that

$$\forall r, V.\quad \mathsf{D}(r, V) = 1 \iff \exists V'.\ (\forall (id, v) \in V. \forall (id', v') \in V'.\ id \neq id') \ \wedge\ r = \rho(V \parallel V').$$

Definition 63 (Individual verifiability)**.** *A voting system $\mathcal{V}$ for a stable counting function ρ is* individually verifiable *against a malicious ballot box if for any adversary $\mathcal{A}$,*

$$\mathbb{P}\left[\mathsf{Exp}^{\mathsf{mb\text{-}verif}}_{\mathcal{A},\mathcal{V}}(\lambda) = 1\right] \text{ is negligible.}$$

As we explained above, for the ideal functionality $\mathcal{F}^P_{\mathsf{V}}$ to entail verifiability, we need P to enforce it. We formalise this condition as follows.

Definition 64 (Verification-friendly predicate)**.** *A predicate P is* verification-friendly *if it ensures that the votes of all voters in $\mathsf{H}_{\mathsf{check}}$ are not removed, reordered, nor modified. That is, if for all L and f such that $P(L, f) = \top$ we have*

$$\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i \in [\![1, |L|]\!] | \exists v.\ L[i] = (id, v)] = [f(j), j = 1 \dots |\mathrm{dom}(f)| | \exists v.\ L[f(j)] = (id, v)]$$

and

$$\forall i, id, v.\ f(i) = (id, v) \implies id \notin \mathsf{H}_{\mathsf{check}}.$$

For instance, the predicates P^{del} and $P^{\emptyset}$ defined in earlier sections satisfy this property, while $P^{\mathsf{del,reorder}}$ and $P^{\mathsf{del,reorder,change}}$ do not, since they allow votes to be reorder. With the additional assumption that voters do not revote, this reordering issue disappear, and $P^{\mathsf{del,reorder}}$ and $P^{\mathsf{del,reorder,change}}$ would be verification-friendly.

For stable counting functions, the following property confirms that the verification-friendliness correctly formalises the intuition that P enforces verifiability.

Lemma 24. *Consider a stable counting function ρ and a verification-friendly predicate P. Let L be a sequence of identities and votes, and f a vote tampering function such that $P(L, f) = \top$. Let $L_{\mathsf{check}} = [(id, v) \in L \mid id \in \mathsf{H}_{\mathsf{check}}]$ be the sublist of L containing only the votes from voters in $\mathsf{H}_{\mathsf{check}}$.*

Then there exists a sequence V_c such that

$$\forall (id, v) \in V_c.\ id \notin \mathsf{H}_{\mathsf{check}}$$

and

$$\rho(\overline{f}(L)) = \rho(L_{\mathsf{check}} \parallel V_c).$$

Proof. This property follows immediately from the definition of $\overline{f}(L)$ and the assumptions on P and ρ. Indeed, the verification-friendliness of P means that f keeps all votes from voters in $\mathsf{H}_{\mathsf{check}}$ in the same order as in L, and does not modify any vote for these voters. Thus $\overline{f}(L)$ contains L_{check} as a sublist (possibly interleaved with other votes), and the remaining elements of $\overline{f}(L)$ are pairs (id, v) such that $id \notin \mathsf{H}_{\mathsf{check}}$. By stability of ρ, permuting the elements of $\overline{f}(L)$ so that L_{check} is upfront does not change the result, which proves the claim. □

We then lift this property to recovery algorithms as follows:

Definition 65 (Verification-friendly recovery)**.** *A* Recover *algorithm is* verification-friendly *if there exists a verification-friendly predicate P such that* Recover *is compatible with P.*

Note that the notion of individual verifiability considered here is a rather basic one, which only requires that votes from voters who verify are counted. In particular it does impose any restrictions on the number of dishonest votes that can be added, or on what happens to the votes of voters who do not verify. A stronger notion of individual verifiability could *e.g.* require that the adversary cannot add more dishonest votes than the number of corrupt voters, to express that the protocol protects against ballot stuffing. This would be done in the game *e.g.* by bounding the size of the list V_c of dishonest votes. A way to further strengthen the property would be to require that the votes of honest voters who do not verify can be removed, but not arbitrarily modified – Belenios provides such guarantees for instance. It would likely be possible to prove that mb-BPRIV implies such stronger notions of verifiability, by considering accordingly a stronger notion of verification-friendliness for predicates.

6.7.2 Results

Armed with these properties, we can now formally state and prove the claim from the beginning of the section.

Theorem 13 (Simulation security implies individual verifiability)**.** *Consider a strongly consistent voting scheme $\mathcal{V}$, for a stable counting function ρ, and let P be a verification-friendly predicate.*

If $\mathcal{V}$ securely implements $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$, then $\mathcal{V}$ is individually verifiable against a dishonest ballot box.

Proof. By contraposition, consider an adversary $\mathcal{A}$ that wins the individual verifiability game $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}verif}}$ with non-negligible probability.

We then construct an adversary $\mathcal{B}$ and an environment $\mathcal{E}$ for the real execution $\mathsf{realexec}(\mathcal{E}||\mathcal{B}||\mathcal{V})$, such that $\mathcal{B}$ cannot be simulated in the eyes of $\mathcal{E}$ by a simulator $\mathcal{S}$ in the ideal execution $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P}(\rho))$. Intuitively, $\mathcal{E}$ will run $\mathcal{A}$ internally, and instruct $\mathcal{B}$ how to tamper with the ballot box following $\mathcal{A}$'s behaviour. In the end, $\mathcal{E}$ will check whether verifiability is broken or not. Since $\mathcal{A}$ wins the verifiability game, in the real execution it will be; however no simulator will be able to break it in the ideal world.

$\mathcal{B}$ and $\mathcal{E}$'s executions is as follows. $\mathcal{E}$ runs $\mathcal{A}$ internally, and starts by asking $\mathcal{H}$ to start the setup phase. During that phase, $\mathcal{B}$ receives from H the public parameters of the election (the key pk, and the public credentials of all voters), and the private credentials of corrupt voters. $\mathcal{B}$ transmits those to $\mathcal{E}$, who in turn transmits them to $\mathcal{A}$. $\mathcal{E}$ then asks $\mathcal{H}$ to start the voting phase, and continues to run $\mathcal{A}$. Whenever $\mathcal{A}$ calls oracle $\mathcal{O}\mathsf{vote}(id, v)$, $\mathcal{E}$ sends $\mathtt{vote}(id, v)$ to $\mathcal{H}$. $\mathcal{B}$ then obtains the corresponding ballots from $\mathcal{H}$, transmits it to $\mathcal{E}$, who returns it to $\mathcal{A}$. During this process, $\mathcal{E}$ records the sequence Voted of identities and votes chosen by $\mathcal{A}$ for voters in $\mathsf{H}_{\mathsf{check}}$. Once $\mathcal{A}$ returns some board BB, $\mathcal{E}$ transmits it to $\mathcal{B}$, and asks $\mathcal{H}$ to end the voting phase. $\mathcal{H}$ then retrieves BB from $\mathcal{B}$, checks its validity, and asks $\mathcal{B}$ for the verification order. $\mathcal{E}$ continues to internally run $\mathcal{A}$, and transmits the order in which it makes $\mathcal{O}\mathsf{verify}$ queries to $\mathcal{B}$. $\mathcal{B}$ returns this order to $\mathcal{H}$, who runs the verifications. If they succeed, $\mathcal{H}$ asks $\mathcal{T}$ to tally BB, and asks $\mathcal{B}$ whether to publish the result. $\mathcal{B}$ agrees to publish it, and $\mathcal{H}$ sends the result r to $\mathcal{E}$. $\mathcal{E}$ runs $\mathsf{D}(r, \mathsf{Voted})$, and outputs 0 if r contains Voted (or is $\perp$), and 1 if it does not. If at any point in its execution $\mathcal{E}$ does not receive the expected messages from $\mathcal{B}$, it returns 0 when asked for a guess. In addition, if $\mathcal{A}$ did not make all voters in $\mathsf{H}_{\mathsf{check}}$ vote (and later verify), $\mathcal{E}$ ends its execution, and answers 0 as a guess.

It is clear that the real execution $\mathsf{realexec}(\mathcal{E}||\mathcal{B}||\mathcal{V})$ follows the execution of game $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}verif}}$, and therefore that $\mathbb{P}(\mathsf{realexec}(\mathcal{E}||\mathcal{B}||\mathcal{V}) = 1) = \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}verif}} = 1)$.

Consider now any simulator $\mathcal{S}$ for the ideal execution $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P}(\rho))$. For $\mathcal{E}$ to output 1 in that execution, it needs to receive a result $r \neq \perp$ such that $\mathsf{D}(r, \mathsf{Voted}) = 0$. At some point in this execution, $\mathcal{S}$ submits a tampering function f to $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$. If f is rejected by P, $\mathcal{E}$ does not get any result, and thus answers 0. The only way for $\mathcal{E}$ to guess 1 is that $P(L, f) = \top$, where L is the list of votes recorded by the functionality. $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ will then compute $r = \rho(\overline{f}(L))$, and $\mathcal{S}$ must accept this result so that $\mathcal{E}$ receives it from $\mathcal{H}'$, otherwise $\mathcal{E}$ answers 0. By Lemma 24, we have $r = \rho(\mathsf{Voted} \parallel V_c)$ for some V_c that does not contain identities in $\mathsf{H}_{\mathsf{check}}$, since Voted is the sequence of votes submitted by $\mathcal{E}$ for voters in $\mathsf{H}_{\mathsf{check}}$. Therefore, $\mathsf{D}(r, \mathsf{Voted}) = 1$, and $\mathcal{E}$ still answers 0.

Hence in the ideal execution, with any possible simulator, $\mathcal{E}$ can never answer 1, while it returns 1 with the non-negligible probability $\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}verif}}_{\mathcal{A},\mathcal{V}} = 1)$ in the real execution with $\mathcal{B}$. Therefore, $\mathcal{B}$ cannot be accurately simulated, and $\mathcal{V}$ does not securely implements $\mathcal{F}^{P}_{\mathsf{v}}(\rho)$, which proves the theorem. □

This result extends to mb-BPRIV thanks to Theorem 11.

Theorem 14 (mb-BPRIV implies individual verifiability)**.** *Consider a strongly consistent voting scheme $\mathcal{V}$, for a stable counting function ρ, and let* Recover *be a verification-friendly algorithm.*

If $\mathcal{V}$ is mb-BPRIV w.r.t. Recover*, then $\mathcal{V}$ is individually verifiable against a dishonest ballot box.*

Proof. This directly follows from the previous theorem regarding the ideal functionality. Indeed, by definition, there exists a verification-friendly predicate P with which Recover is compatible. Thus by Theorem 11, $\mathcal{V}$ securely implements $\mathcal{F}^{P}_{\mathsf{v}}(\rho)$, and thus by the previous theorem, $\mathcal{V}$ is individually verifiable. □

6.8 Entropy

In the parametric ideal functionality we defined, the simulator (*i.e.* the ideal adversary) is allowed to tamper with the votes of honest voters, via a tampering function. As explained before, allowing such tampering even in the ideal world is necessary when considering a malicious ballot box. Indeed, in a real system, the ballot box can typically remove the ballots of voters provided they do not verify their vote. Therefore, if one wants to consider the more complex, but realistic, scenario where not everyone performs the verification steps, the adversary must be given at least some power over the honest votes even in the ideal world.

Designing the ideal functionality in a parametric way as we have done has the advantage of letting us characterise precisely in what way the adversary can tamper with the votes in the scheme we consider, which, again, is mandatory if we want to model a dishonest ballot box. The downside, however, is that even against the ideal system, the adversary is not powerless. That makes it somewhat difficult to understand to what extent exactly a given instance of our ideal functionality guarantees privacy.

For instance, it is pretty clear that the very restrictive functionality $\mathcal{F}^{\emptyset}_{\mathsf{v}}$ (from Section 6.5.2) provides a high level of privacy: recall that this functionality requires that, regardless of which voters verify, no honest votes can ever be removed, reordered, or modified in any way. The adversary can only add dishonest votes, and see the result. At the other end of the scope, consider the most permissive functionality $\mathcal{F}^{\top}_{\mathsf{v}}$, associated with the predicate that is always true. This functionality allows the adversary to tamper with the votes in any way he wishes, no matter which voters verify. Here it is quite easy to see that this functionality does not provide any privacy: a simulator who wishes to know some voter Alice's vote can simply remove every vote but hers, and of course the result lets him learn Alice's vote. Between those two extremes however, it is much less clear what exactly the adversary can learn about the votes. For example, consider a functionality such as $\mathcal{F}^{\mathsf{del,reorder}}_{\mathsf{v}}$ (Section 6.5.2) where the adversary cannot change the

value of any honest vote, and in addition cannot remove the votes from the voters who verify[16] Say the adversary is trying to learn Alice's vote, and Alice has verified her vote. There the adversary can no longer remove all votes except hers. However he can remove all votes from voters who did not verify. The result then only contains the votes of voters in $\mathsf{H}_{\mathsf{check}}$. This does not let the adversary learn Alice's vote, but intuitively it leaks *some* information on her vote, or more precisely, it leaks *more information* than if no votes at all were removed. It is not quite clear a priori what privacy Alice gets in this setting.

In the same vein, it is not trivial in our model to understand precisely how different ideal functionalities compare. Consider for instance functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$ (Section 6.5.2). It is similar to $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$ except the adversary can now also modify the value of the votes from honest voters, provided they do not verify (while these can only be removed, but not changed, in $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$). It seems intuitively clear that the adversary can learn at least as much information on Alice's vote in $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$ as in $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$, as the functionality is more permissive. However, it is not clear at all *how much more information* about this vote he can get. In fact, it is not even obvious that he can obtain *strictly* more information: how exactly does the additional power he gets compared to $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$ help him?

One natural tool to try to quantify the information available to the adversary, and thus to compare the ideal functionalities, is the notion of *entropy*. The question then becomes: how much entropy is left in Alice's vote (or any information the adversary is trying to learn) once the adversary sees the result? Or more precisely: what is the minimal value of this entropy, when considering all possible simulators against a given ideal functionality?

In this final section, we propose to explore how the notion of entropy can be used to get a better insight on the privacy our ideal functionality provide, by trying to compare $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$.

Bernhard, Cortier, Pereira and Warinschi introduced in [27] a framework to formalise these questions. In what follows, we present an adaptation this framework to our model, and establish a general result to help compare the entropy for our family of ideal functionalities. We then answer at least partially the previous question regarding the comparison of $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$. Finally, we compute (by numerical simulation) the entropy in various examples, and discuss our observations.

6.8.1 Entropy for voting protocols

Let us first recall some standard notions and notations regarding entropy.

Entropy

Entropy is a concept from information theory, that intuitively intends to measure the quantity of information contained in a random variable. A well-established notion of entropy is Rényi entropy, which constitutes a family of entropy functions parametrised by a real number α, defined as follows:

Definition 66 (Rényi entropy)**.** *Let* $\mathbf{X}$ *be a random variable with values in* $\mathcal{X}$*. For* $\alpha \in [0, \infty[\setminus\{1\}$*, the Rényi entropy with parameter* α *of* $\mathbf{X}$ *is*

$$\mathbb{H}_\alpha(\mathbf{X}) = \frac{1}{1-\alpha} \log \left(\sum_{x \in \mathcal{X}} \mathbb{P}(\mathbf{X} = x)^\alpha \right).$$

[16] Ignoring the "reordering" part, which allows the adversary to change the order of Alice's votes in case Alice submitted several. For simplicity we consider in this section the case where voters do not revote.

We extend the definition to the cases of $\alpha = 1$ and $\alpha = \infty$ by passing to the limit.

Some particular values of α yield some well-known entropy functions:

- For $\alpha = 0$, Hartley entropy, which measures the number of possible values for $\mathbf{X}$:

$$\mathbb{H}_0(\mathbf{X}) = \log(|\mathcal{X}|).$$

- For $\alpha = 1$, Shannon entropy intuitively measures the average over the value x of $\mathcal{X}$ of the size in bits needed to store the information that $\mathbf{X} = x$:

$$\mathbb{H}_1(\mathbf{X}) = -\sum_{x \in \mathcal{X}} \mathbb{P}(\mathbf{X} = x) \log(\mathbb{P}(\mathbf{X} = x)).$$

- For $k = \infty$, min-entropy measures the probability that the "best guess" as to the value of $\mathbf{X}$, *i.e.* the x that maximises $\mathbb{P}(\mathbf{X} = x)$, is correct:

$$\mathbb{H}_\infty(\mathbf{X}) = -\log(\max_{x \in \mathcal{X}}(\mathbb{P}(\mathbf{X} = x))).$$

For a given Rényi entropy function $\mathbb{H}_\alpha$, if $\mathbf{X}$ is a random variable and A some event, we denote $\mathbb{H}_\alpha(\mathbf{X}|A)$ the conditional entropy of $\mathbf{X}$ given A, which is defined just as $\mathbb{H}(\mathbf{X})$, except the probability $\mathbb{P}(\mathbf{X} = x)$ is replaced with the conditional probability $\mathbb{P}(\mathbf{X} = x|A)$.

We will next need to define the *conditional entropy of* $\mathbf{X}$ *given* $\mathbf{Y}$, which intuitively measures the entropy left in $\mathbf{X}$ once the value of $\mathbf{Y}$ is known. This does not trivially follow from $\mathbb{H}_\alpha(\mathbf{X}|A)$, since "the value of $\mathbf{Y}$" is not an event: for any y, "$\mathbf{Y} = y$" is and the previous definition applies to $\mathcal{H}_\alpha(\mathbf{X}|\mathbf{Y} = y)$, but we need to define how to combine these over all values of y. In [27], the authors consider three possible ways of doing this, which we reproduce here.

Definition 67 (Conditional entropy)**.** *For an entropy function $\mathbb{H}$, the conditional entropy of $\mathbf{X}$ given $\mathbf{Y}$ is the expected entropy of $\mathbf{X}$ given the value of $\mathbf{Y}$, over all possible values of $\mathbf{Y}$:*

$$\mathbb{H}(\mathbf{X}|\mathbf{Y}) = \mathop{\mathbb{E}}_{y \in \mathcal{Y}}(\mathbb{H}(\mathbf{X}|\mathbf{Y} = y)).$$

Definition 68 (Minimal entropy)**.** *For an entropy function $\mathbb{H}$, the minimal entropy of $\mathbf{X}$ given $\mathbf{Y}$ is the minimal entropy of $\mathbf{X}$ given the value of $\mathbf{Y}$, over all possible values of $\mathbf{Y}$:*

$$\mathbb{H}^\perp(\mathbf{X}|\mathbf{Y}) = \min_{y \in \mathcal{Y}}(\mathbb{H}(\mathbf{X}|\mathbf{Y} = y)).$$

Definition 69 (Average min-entropy)**.** *In the case of the min-entropy $\mathbb{H}_\infty$, we consider the average min-entropy of $\mathbf{X}$ given $\mathbf{Y}$, defined*

$$\overline{\mathbb{H}}_\infty(\mathbf{X}|\mathbf{Y}) = -\log\left(\mathop{\mathbb{E}}_{y \in \mathcal{Y}}\left(2^{-\mathbb{H}_\infty(\mathbf{X}|\mathbf{Y}=y)}\right)\right),$$

i.e.

$$\overline{\mathbb{H}}_\infty(\mathbf{X}|\mathbf{Y}) = -\log\left(\mathop{\mathbb{E}}_{y \in \mathcal{Y}} \max_{x \in \mathcal{X}}(\mathbb{P}(\mathbf{X} = x|\mathbf{Y} = y))\right).$$

[27] then considers a general notion of conditional entropy, which all previous examples are instances of.

Definition 70 (Conditional entropy)**.** *A conditional entropy measure is a function* $\mathbb{H}$ *mapping two random variables* $\mathbf{X}$*,* $\mathbf{Y}$ *to a positive real number such that*

- *If* $\mathbf{X}$ *can be computed as a probabilistic function of* $\mathbf{Y}$*, then* $\mathbb{H}(\mathbf{X}|\mathbf{Y}) = 0$*.*
- *If* $\mathbf{Y}$*,* $\mathbf{Y}'$ *are two random variables such that* $\mathbf{Y}'$ *can be computed as a function of* $\mathbf{Y}$*, then for any* $\mathbf{X}$*,* $\mathbb{H}(\mathbf{X}|\mathbf{Y}') \geq \mathbb{H}(\mathbf{X}|\mathbf{Y})$*.*

In the following we consider an arbitrary, fixed conditional entropy $\mathbb{H}$.

Execution model and view of the adversary

We now present the model we use to characterise the entropy of the votes from the point of view of the adversary, which is inspired from [27].

Consider the ideal functionality $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$, for a given predicate P and counting function ρ. Let $\mathcal{E}$ be an environment, $\mathcal{S}$ a simulator, and consider the ideal execution of $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ with $\mathcal{E}$ and $\mathcal{S}$ $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P})$.

In this execution, we denote $\mathbf{V}$ the random variable whose value is the list of votes chosen by $\mathcal{E}$ for the honest voters (in a fixed order). We also denote V the set of possible votes: $\mathbf{V}$ thus has values in $\mathsf{V}^{|\mathsf{H}|}$ (recall that H is the set of honest voters).

We also call *view* of $\mathcal{S}$, and denote $\mathsf{View}(\mathcal{S}, \mathcal{E}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))$, the sequence of the random coins used by $\mathcal{S}$ and all messages received by $\mathcal{S}$ during the execution. $\mathsf{View}(\mathcal{S}, \mathcal{E}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))$ is a random variable, that depends on (*i.e.* can be computed as a function of) the random coins of $\mathcal{E}$ and $\mathcal{S}$. Notably, it contains the result (or absence of result) of the election, that is sent to $\mathcal{S}$ by the functionality during the execution.

For the purposes of computing entropy on the votes, they need to be random variables that follow a certain probability distribution, intuitively representing the information known *a priori* on the voters' intentions. We make a few assumptions here, for simplicity. First, we will consider that voters vote only once, and that all votes follow the same distribution, and are independent. While not necessarily always accurate in real life – we might know *a priori* that a certain group of voters have a tendency to vote the same for instance – this seems like a reasonable assumption for a first look at how entropy can be useful in our setting. In addition, we will also assume that the environment is not deliberately helping the adversary/simulator by sending him information on the votes. By doing so, we model the case where all honest voters are trying to keep their votes secret. Again, this may not reflect all real life scenarios – in a more complete study we might for instance want to model the case where some voters, while honest, deliberately make their choice public for instance – but is a reasonable assumption to simplify our study.

Formally, we now consider a restricted class of environments $\mathcal{E}_{\mathsf{D}}$, parametrised by a probability distribution D on votes. $\mathcal{E}_{\mathsf{D}}$ tells $\mathcal{H}'$ to start the setup and then voting phases, then draws the vote of each of the honest voters (in a fixed order) independently according to D, and submits all these votes to the functionality (through $\mathcal{H}'$). $\mathcal{E}_{\mathsf{D}}$ then tells $\mathcal{H}'$ to end the voting phase and start the tallying, and sends no further messages to any of the other entities.

We also restrict the class of ideal functionalities $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ we consider, by requiring that for any L, f, $P(L, f)$ can be computed (efficiently) using only the list of identities in L, and not L itself. This way, in an execution, $P(L, f)$ can be computed from the view of $\mathcal{S}$. We denote $\mathcal{P}$ the set of such predicates P. This class of predicates covers all examples described earlier: the restriction is intuitively that whether a voter's vote can be modified or not does not depend the exact value of that vote, but only on the voter's identity (typically, on whether that voter

checks or not). Here again, while it might be interesting to investigate cases where the adversary is able to tamper with a vote only for certain values of that vote, this seems like a reasonable assumption for a first study.

It is clear that with such environments and functionalities, the view $\mathsf{View}(\mathcal{S}, \mathcal{E}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))$ can be computed from only the random coins of $\mathcal{S}$ and the result of the election.

Finally we split the simulators we will consider into several classes. For a fixed $\gamma \in [\![0, |\mathsf{H}|]\!]$, we denote S_γ the set of all simulators $\mathcal{S}$ that keep exactly γ votes unmodified. That is, simulators $\mathcal{S}$ such that in any execution, if L is the list of votes recorded by $\mathcal{F}_{\mathsf{v}}^{P}$, $\mathcal{S}$ returns a tampering function f such that $|\{i \in \mathrm{dom}(f) \mid f[i] \in \mathbb{N}\}| = \gamma$.

Entropy for voting systems

Following [27], we consider the following notion of *target function* that characterises what information the adversary is trying to learn on the votes.

Definition 71 (Target function)**.** *We call* target function *any function* $\mathbf{T} : \mathsf{V}^{|\mathsf{H}|} \to \mathcal{T}$*, for some set* $\mathcal{T}$*. Such a function will represent the information the adversary is trying to get on the list* $\mathbf{V}$ *of honest votes.*

We can now define the entropy of the target from the point of view of the best possible simulator as follows.

Definition 72 (Entropy on the target function)**.** *Consider a probability distribution on votes* D*, a target function* $\mathbf{T}$*, an ideal functionality* $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$*, a number of voters* $\gamma \in [\![0, |\mathsf{H}|]\!]$*. The entropy of* $\mathcal{T}(\mathcal{V})$ *when keeping* γ *votes is*

$$\mathbb{H}_{\mathsf{D},\gamma}^{P,\rho}(\mathbf{T}(\mathbf{V})) = \min_{\mathcal{S} \in S_\gamma} \mathbb{H}(\mathbf{T}(\mathbf{V}) | \mathsf{View}(\mathcal{S}, \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))).$$

This notion is what we will use to quantify the privacy of voters. For instance, if we consider a target function that picks the vote from one specific voter Alice from the list, it measures the minimum entropy that is left in Alice's vote, given the view of any possible simulator (that leaves γ votes untouched). That is, intuitively, how much uncertainty regarding their votes the best adversary still has. We can for instance compare the value of this entropy for different functionalities, or different result functions.

6.8.2 Comparing entropies

General results

With this notion of entropy, we can formally prove the intuitive claim that a more permissive functionality, *i.e.* one with a more permissive predicate, that gives more power to the simulator, leaks more information about the votes. That is, the entropy left in any target given the result is lower for the more permissive functionality.

Theorem 15. *Consider a counting function* ρ*, and two predicates* P*,* P' *in* $\mathcal{P}$*, such that* P *is more permissive than* P'*,* i.e. $\forall L, f.\ P'(L, f) \Rightarrow P(L, f)$*. Recall that* $\mathbf{V}$ *is the random variable consisting in the list of all honest votes. Then*

$$\forall \mathbf{T}.\ \forall \gamma.\ \forall \mathsf{D}.\ \mathbb{H}_{\mathsf{D},\gamma}^{P,\rho}(\mathbf{T}(\mathbf{V})) \leq \mathbb{H}_{\mathsf{D},\gamma}^{P,\rho}(\mathbf{T}(\mathbf{V})).$$

Proof. Let $\mathbf{T}$ be a target function, let $\gamma \in [\![0, |\mathsf{H}|]\!]$, and let D be a probability distribution on votes. Let $\mathcal{S}' \in S_\gamma$ be a simulator for $\mathcal{F}_\mathsf{v}^{P'}(\rho)$.

We construct a simulator $\mathcal{S} \in S_\gamma$ for $\mathcal{F}_\mathsf{v}^{P}(\rho)$ as follows. $\mathcal{S}$ internally runs $\mathcal{S}'$, acting as an interface and forwarding the messages between $\mathcal{S}'$ and the other entities, until the point where $\mathcal{S}'$ wishes to send a tampering function f to the functionality. During the execution, $\mathcal{S}$ records in a list I the identities id for which it receives an acknowledgement $\texttt{ack}(id)$ from $\mathcal{H}'$.

At that point, let L be the list of identities and votes kept by the functionality $\mathcal{F}_\mathsf{v}^{P}(\rho)$ in the execution $\mathsf{idealexec}(\mathcal{E}_\mathsf{D}||\mathcal{S}||\mathcal{F}_\mathsf{v}^{P}(\rho))$. Since, up to this point in the execution, $\mathcal{F}_\mathsf{v}^{P'}(\rho)$ and $\mathcal{F}_\mathsf{v}^{P}(\rho)$ are identical, $\mathcal{S}$ has accurately simulated the execution of $\mathcal{S}'$ in $\mathsf{idealexec}(\mathcal{E}_\mathsf{D}||\mathcal{S}'||\mathcal{F}_\mathsf{v}^{P'}(\rho))$. Thus L is also the list recorded by $\mathcal{F}_\mathsf{v}^{P'}(\rho)$ in that execution.

$\mathcal{S}$ then computes $P'(L, f)$. By assumption, $\mathcal{S}$ can compute this value using only the list of identities in L, *i.e.* I.

If $P'(L, f) = \bot$, $\mathcal{S}$ sends `no tally` to $\mathcal{S}'$, and stops. Otherwise, $\mathcal{S}$ sends f to $\mathcal{F}_\mathsf{v}^{P}(\rho)$. Since P is more permissive than P', $P(L, f) = \top$. $\mathcal{F}_\mathsf{v}^{P}(\rho)$ thus accepts f, computes the result $r = \rho(\overline{f}(L))$, and sends it to $\mathcal{S}$. $\mathcal{S}$ forwards this message to $\mathcal{S}'$, and answers to the functionality what $\mathcal{S}'$ answers.

It is clear that the simulated $\mathcal{S}'$ sees the result if and only if $\mathcal{S}'$ actually sees it in the corresponding execution of $\mathsf{idealexec}(\mathcal{E}_\mathsf{D}||\mathcal{S}'||\mathcal{F}_\mathsf{v}^{P'}(\rho))$, and that in that case the result $\mathcal{S}'$ sees is the same in both executions. In other words, the view of the simulated $\mathcal{S}'$ is the same as the view $\mathsf{View}(\mathcal{S}', \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P'}(\rho))$ that $\mathcal{S}'$ has in $\mathsf{idealexec}(\mathcal{E}_\mathsf{D}||\mathcal{S}'||\mathcal{F}_\mathsf{v}^{P'}(\rho))$.

Since $\mathcal{S}$ was able to simulate such an $\mathcal{S}'$, the view $\mathsf{View}(\mathcal{S}', \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P'}(\rho))$ of $\mathcal{S}'$ can therefore be computed from the view $\mathsf{View}(\mathcal{S}, \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P}(\rho))$ of $\mathcal{S}$. Thus, by the assumptions on the conditional entropy $\mathbb{H}$, we have

$$\mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}', \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P'}(\rho))) \geq \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}, \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P}(\rho))).$$

Hence,

$$\forall \mathcal{S}' \in S_\gamma.\ \exists \mathcal{S} \in S_\gamma.\ \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}', \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P'}(\rho))) \geq \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}, \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P}(\rho))),$$

which implies

$$\min_{\mathcal{S}' \in S_\gamma} \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}', \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P'}(\rho))) \geq \min_{\mathcal{S} \in S_\gamma} \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}, \mathcal{E}_\mathsf{D}, \mathcal{F}_\mathsf{v}^{P}(\rho))),$$

i.e.

$$\mathbb{H}_{\mathsf{D},\gamma}^{P',\rho}(\mathbf{T}(\mathbf{V})) \geq \mathbb{H}_{\mathsf{D},\gamma}^{P,\rho}(\mathbf{T}(\mathbf{V})).$$

□

Applied to the functionalities $\mathcal{F}_\mathsf{v}^{\mathsf{del},\mathsf{reorder}}$ and $\mathcal{F}_\mathsf{v}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$, this theorem proves the intuition that the former provides at least as much privacy as the latter to voters. As discussed previously, the natural question is then: does it actually provide strictly more privacy or not? Thanks to the following result, that generalises Theorem 15, we can partially answer this question.

Theorem 16. *Consider a counting function ρ, and two predicates P, P' in $\mathcal{P}$. Assume there is a way of turning a vote tampering function accepted by P into a tampering function accepted by P' without losing information. That is, there exist polynomial algorithms A and B such that*

- *A turns a function f accepted by P into a function $A(f)$ accepted by P':*

$$\forall L, f.\ P(L, f) \implies P'(L, A(f)).$$

- *In addition, the function returned by A always contains the same number of unchanged votes as its input:*

$$\forall f.\ |\{i \in \mathrm{dom}(A(f)) \mid A(f)(i) \in \mathbb{N}\}| = |\{i \in \mathrm{dom}(f) \mid f(i) \in \mathbb{N}\}|.$$

- *B is able to compute the tally obtained when applying the tampering f, from the one obtained when applying $A(f)$. That is*

$$\forall L, f.\ \rho(\overline{f}(L)) = B(f, \rho(\overline{A(f)}(L))).$$

Then $\mathcal{F}_{\mathsf{v}}^{P'}$ leaks at least as much information as $\mathcal{F}_{\mathsf{v}}^{P}$ on any target:

$$\forall \mathbf{T}.\ \forall \gamma.\ \forall \mathsf{D}.\ \mathbb{H}_{\mathsf{D},\gamma}^{P',\rho}(\mathbf{T}(\mathbf{V})) \leq \mathbb{H}_{\mathsf{D},\gamma}^{P,\rho}(\mathbf{T}(\mathbf{V})).$$

Proof. Let $\mathbf{T}$ be a target function, let $\gamma \in [\![0, |\mathsf{H}|]\!]$, and let D be a probability distribution on votes. Let $\mathcal{S} \in S_\gamma$ be a simulator for $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$.

This proof follows a similar intuition to the previous one. As before, we construct a simulator $\mathcal{S}' \in S_\gamma$ for $\mathcal{F}_{\mathsf{v}}^{P'}(\rho)$, that will internally run $\mathcal{S}$ in order to internally compute its view, as follows. $\mathcal{S}'$ internally runs $\mathcal{S}$, acting as an interface and forwarding the messages between it and the other entities, until the point where $\mathcal{S}$ wishes to send tampering function f. During the execution, $\mathcal{S}'$ records the list I the identities for which it receives an acknowledgement.

At that point, let L be the list of identities and votes kept by the functionality $\mathcal{F}_{\mathsf{v}}^{P'}(\rho)$ in the execution $\mathsf{idealexec}(\mathcal{E}_{\mathsf{D}}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{P'}(\rho))$. As noted before, up to this point in the execution, $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ and $\mathcal{F}_{\mathsf{v}}^{P'}(\rho)$ are identical, and thus $\mathcal{S}'$ has accurately simulated the execution of $\mathcal{S}$ in $\mathsf{idealexec}(\mathcal{E}_{\mathsf{D}}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P}(\rho))$. Thus L is also the list recorded by $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$ in that execution.

$\mathcal{S}'$ then computes $P(L, f)$, which it can do using only f and I by assumption. If $P(L, f) = \bot$, $\mathcal{S}'$ sends `no tally` to $\mathcal{S}$, and halts. Otherwise, $\mathcal{S}'$ computes $f' = A(f)$, and sends it to $\mathcal{F}_{\mathsf{v}}^{P'}(\rho)$. Since $P(L, f) = \top$, by assumption on A, $P'(L, f') = \top$. $\mathcal{F}_{\mathsf{v}}^{P'}(\rho)$ thus accepts f', computes the result $r = \rho(\overline{f'}(L))$, and sends it to $\mathcal{S}'$. $\mathcal{S}'$ then computes $r' = B(f, r)$. By assumption on B, we have $r' = \rho(\overline{f}(L))$.

$\mathcal{S}'$ then sends r' to $\mathcal{S}$, and forwards its response to the functionality.

It is clear that the simulated $\mathcal{S}$ sees the result if and only if $\mathcal{S}$ actually sees it in the corresponding execution of $\mathsf{idealexec}(\mathcal{E}_{\mathsf{D}}||S||\mathcal{F}_{\mathsf{v}}^{f}(\rho))$, and in that case sees the same result. In other words, the view of the simulated $\mathcal{S}$ is the same as the view $\mathsf{View}(\mathcal{S}, \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))$ that $\mathcal{S}$ has in $\mathsf{idealexec}(\mathcal{E}_{\mathsf{D}}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{P}(\rho))$.

Since $\mathcal{S}'$ was able to simulate such an $\mathcal{S}$, the view $\mathsf{View}(\mathcal{S}, \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))$ of $\mathcal{S}$ can therefore be computed from the view $\mathsf{View}(\mathcal{S}', \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P'}(\rho))$ of $\mathcal{S}'$. Thus, by the assumptions on the conditional entropy $\mathbb{H}$, we have

$$\mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}, \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))) \geq \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}', \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P'}(\rho))).$$

By assumption on A, f' contains the same number of unmodified votes as f, which means that $\mathcal{S}' \in S_\gamma$. Hence,

$$\forall \mathcal{S} \in S_\gamma.\ \exists \mathcal{S}' \in S_\gamma.\ \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}, \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))) \geq \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}', \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P'}(\rho))),$$

which implies

$$\min_{\mathcal{S} \in S_\gamma} \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}, \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P}(\rho))) \geq \min_{\mathcal{S}' \in S_\gamma} \mathbb{H}(\mathbf{T}(\mathbf{V})|\mathsf{View}(\mathcal{S}', \mathcal{E}_{\mathsf{D}}, \mathcal{F}_{\mathsf{v}}^{P'}(\rho))),$$

i.e.

$$\mathbb{H}_{\mathsf{D},\gamma}^{P,\rho}(\mathbf{T}(\mathbf{V})) \geq \mathbb{H}_{\mathsf{D},\gamma}^{P',\rho}(\mathbf{T}(\mathbf{V})).$$

□

We can note that Theorem 15 is a particular case of Theorem 16, where A and B are simply the identity functions: $A(f) = f$ and $B(f, r) = r$.

Comparing $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$

Consider now the application of Theorem 16 to predicates $P^{\mathsf{del,reorder,change}}$ and $P^{\mathsf{del,reorder}}$. Assume the counting function ρ has the partial tallying property, defined in earlier chapters.

We can in that case define algorithms A and B that satisfy the requirements of the theorem. $A(f)$ removes from f all votes cast by the simulator in the name of honest voters, since those are not allowed by $P^{\mathsf{del,reorder}}$. That is, $A(f)$ returns a function $f' = \lambda i.L'[i]$, where

$$L' = [f(i), i = 1..|\mathrm{dom}(f)| \mid f(i) \in \mathbb{N} \ \vee \ \exists id \notin \mathsf{H}.\exists v \in \mathsf{V}.\ f(i) = (id, v).$$

$B(f, r)$ then computes $r * \rho(L'')$, where L'' is the list of votes cast by the simulator for honest voters in f, *i.e.*

$$L'' = [f(i), i = 1 \ldots |\mathrm{dom}(f)| \mid \exists id \in \mathsf{H}.\exists v \in \mathsf{V}.\ f(i) = (id, v).$$

That is, B adds back to r the tally of all votes A discarded.

With these A and B, it is clear that the conditions of Theorem 16 are fulfilled. This proves that the entropy for $P^{\mathsf{del,reorder}}$ is lower than the entropy for $P^{\mathsf{del,reorder,change}}$.

Since, on the other hand, $P^{\mathsf{del,reorder,change}}$ is clearly more permissive than $P^{\mathsf{del,reorder}}$, by Theorem 15, the converse inequality also holds. Together, these two observations thus establish that these entropies are equal, as formally stated in the following theorem:

Theorem 17 ($\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$ have the same entropy)**.** *For any target function, any γ, any distribution* D*,* $\mathbf{V}$ *being the list of votes, we have*

$$\mathbb{H}_{\mathsf{D},\gamma}^{P^{\mathsf{del,reorder}},\rho}(\mathbf{T}(\mathbf{V})) = \mathbb{H}_{\mathsf{D},\gamma}^{P^{\mathsf{del,reorder,change}},\rho}(\mathbf{T}(\mathbf{V})).$$

That is, the most information a simulator that leaves exactly γ votes unchanged can obtain on $\mathbf{T}(\mathbf{V})$ is the same regardless of whether this simulator is allowed to change the votes of voters that do not verify, or only to drop them.

That result may be surprising, as it intuitively seems that the adversary has much more power in the first case. The confusion stems from the distinction between privacy and verifiability.

The result does not mean that the two functionalities $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$ are equivalent. $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$ describes a system which guarantees that even if I do not perform any verification, my vote cannot be modified, but only removed. As our case study showed earlier, this can be realised in practice *e.g.* by signing the ballots as is done in Belenios. This gives strictly stronger guarantees from the point of view of verifiability that $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$. Although, as proven in the previous section, both functionalities guarantee individual verifiability since we consider here a case without revote, they are not the same. Indeed, for $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$, we could have proven the stronger version of verifiability described earlier, that prevents changing votes even when voters do not check. $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$ on the other hand does not guarantee this stronger verifiability.

However, this result means that, purely from the point of view of privacy, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$ does not give stronger guarantees than $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$, at least in the case of a counting function with

partial tallying. For such counting functions, ensuring that votes cannot be changed even for voters who do not verify (as is done *e.g.* in Belenios) does not provide more privacy than letting an adversary change these votes (as is the case *e.g.* in Helios).

A natural question is then: what if the counting function does not have the partial tallying property? We do not have a general answer to this question, however in the next subsection we investigate it on a few examples.

6.8.3 Experiments

A typical example of a function that does not have the partial tallying property is the winner function. That is, the case where the result is simply the name of the winner, *i.e.* the candidate that received the majority of votes, without any indication regarding the number of votes for each candidate. Indeed, even if we know that candidate 1 got the majority of votes among the first half of voters, and that candidate 2 got the majority among the other half, that does not let us compute who is the winner when counting all the votes.

Intuitively, in that case, it seems that being able to change votes may leak more information to the adversary. While just seeing the winner only tells him that that candidate has more than 50% of votes, changing some of the votes may give information about the margin by which the winner wins. For instance, the intuition seems to indicate that turning 10% of the votes from voters who did not check into votes for the losing candidate would let the adversary learn whether the winner actually had more than 60% of votes; while simply removing these votes would not.

Even in simple cases, computing an exact expression for the entropy can be difficult. It indeed requires to compute the probability distribution of the target function given the result of the election, which seems to often lead to quite complex combinatorics problems, that we did not study here. In what follows, we instead perform numerical simulations to approximate the entropy in some examples involving the winner function, to get some insight on how changing votes influences privacy in that case.

Setting

Concretely, we consider the following simplified setting:

- All voters are honest, *i.e.* the adversary does not control any voter. A number α of voters verify their vote, and β voters do not. When considering the random variable $\mathbf{V}$, we will order them, first numbering the voters who verify from 1 to α, and then the others from $\alpha + 1$ to $n = \alpha + \beta$.
- Votes are vectors of d integers (in our examples we use $d = 2$, 3 or 4). Typically, this could model an election with d candidates: a vote for candidate i would then be a vector v such that $v_i = 1$ and $v_j = 0$ for $j \neq i$.
- We consider two result functions:
  - the sum: summing all votes component by component, *i.e.*

$$\rho_{\mathsf{sum}}(v_1, \ldots, v_k) = (\sum_i v_{i,1}, \ldots, \sum_i v_{i,d})$$

    If votes are for one candidate as described above, the sum gives the number of votes for each candidate.

– the winner: the index of the candidate with the most votes, *i.e.*

$$\rho_{\mathsf{winner}}(v_1, \ldots, v_k) = \mathrm{argmax}_l(\sum_i v_{i,l})$$

or, equivalently, the corresponding vote, *i.e.* the vector with a 1 at that index.

- We consider two examples of target functions:
  - The value of a fixed voter's vote, say the first voter, *i.e.* $\tau_{\mathsf{first}}(v_1, \ldots, v_n) = v_1$. The order specified above means that this voter verifies.
  - The boolean "are at least a proportion p of the votes from the α voters who verify for the first candidate?", for a fixed $p \in [0, 1]$, *i.e.*

$$\tau_p(v_1, \ldots, v_n) = 1 \text{ if } \sum_{1 \leq i \leq \alpha} v_{i,1} \geq p\alpha, \text{ and 0 otherwise.}$$

This function may seem strange, as it can trivially be observed from the result when the counting function is ρ_{sum}. However, it cannot when the function is ρ_{winner}, and in that case it formalises the intuition about learning information on the margin exposed above.

The authors in [27] argue that the most relevant notion of entropy to study is the average min-entropy. We did not dispute this claim, and in most cases we used this notion. However in some cases we also had a look at the conditional Shannon entropy.

In addition we will only consider deterministic simulators. Hence, their view will only depend on the tally they receive (which of course depends on the votes drawn at random by the environment). We therefore compute an approximation of $\mathbb{H}(\mathbf{T}(\mathbf{V}) \mid \rho(\mathbf{V}))$ for a given simulator, V being a list of n votes, each drawn from the same distribution. To do this, we wrote a small program that simulates many elections, drawing votes at random (following a specified distribution). We fix a simulator, for which we wish to compute the entropy. Each time, the program applies the tampering function that simulator would propose to the functionality. It then records the value of the target $\mathbf{T}(\mathbf{V})$ and the result $\rho(\mathbf{V})$. The entropy is then computed using the expressions from the beginning of this section, using frequencies from the experiments rather than probabilities. More experiments are run until the computed value stabilises (*i.e.*, until its variations pass under an arbitrary threshold).

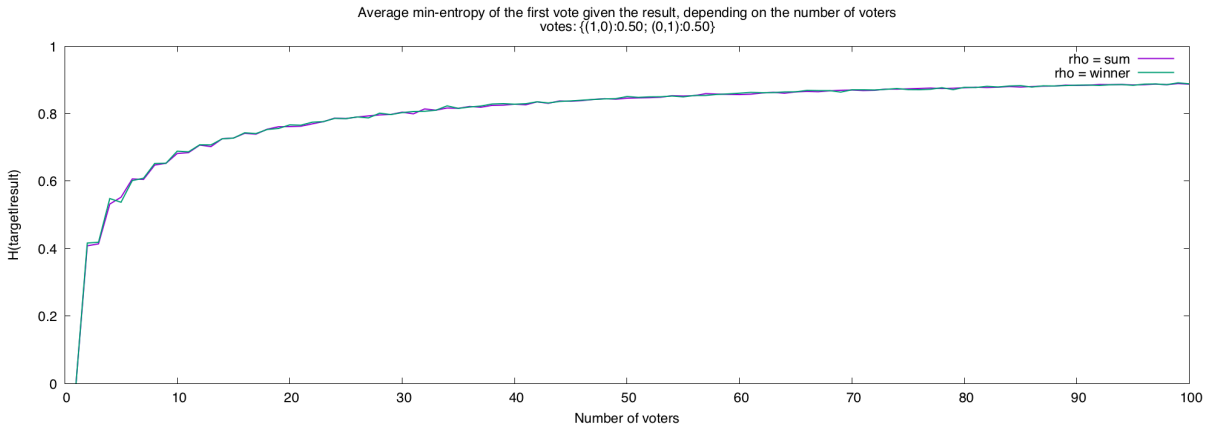
We are of course aware that this is in no way a proof, and gives no guarantee that we compute the correct value for the entropy. This experiment was only a means to try to get a better intuition on what happens in the scenario we considered. However, we still report these results, as they lead to somewhat interesting observations.

Does ρ_{sum} leak more information than ρ_{winner}?

First we consider the case of an adversary (simulator) that does not try to change any votes, but deletes all β votes from the voters who do not verify, as is allowed by both $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$. Hence the result is computed only on α votes that the adversary does not control. Intuitively, removing these β votes is always a better choice for an adversary than leaving them, as the targets we considered only count the α votes from those who verify.

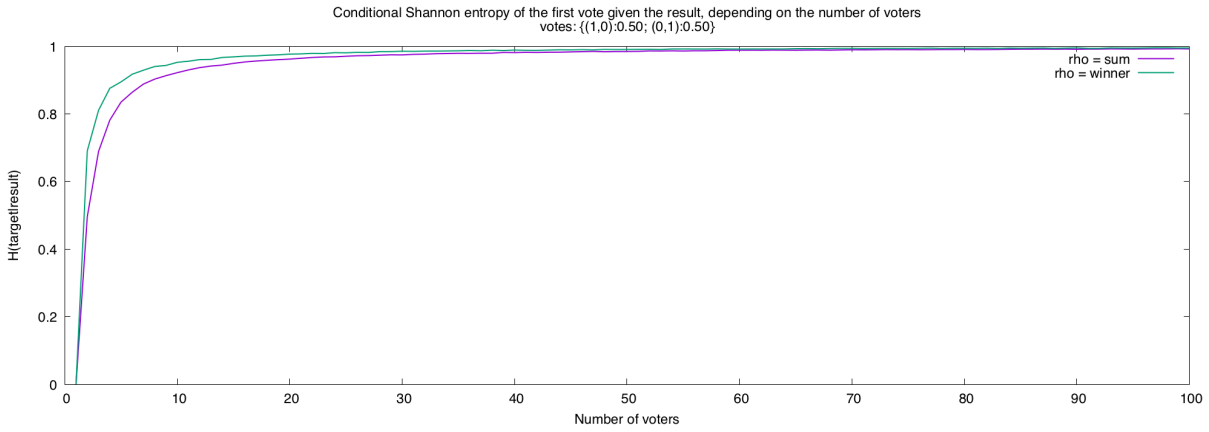
For this adversary, we investigate the difference between the counting functions ρ_{sum} and ρ_{winner}. Intuitively, the first seems to give strictly more information to the adversary than the second. However, we observe that this is not always true.

Consider first that the target is the first vote τ_{first}, and a simple setting where there are two candidates ($d = 2$), and the environment makes voters pick at random among them following a uniform distribution. Of course, the amount of information learnt by the adversary from the result depends on the number α of voters who verify (since only their votes are counted): the more voters there are, the more a specific vote is hidden in the result. Plotting the average min-entropy of $\tau_{\mathsf{first}}(\mathbf{V})$ knowing the result yields the following graph.



Contrary to what one would expect, it appears that the average min-entropy is the same for the sum and winner functions ρ_{sum} and ρ_{winner}. Doing more experiments, with other numbers of candidates and other distributions produces the same observations.

However, computing the conditional Shannon entropy rather than the average min-entropy yields a different result, as displayed on the following graph.



Here we see a difference between the two counting function: as expected, the winner function leaks less information, leading to a higher entropy.

While this may be surprising, it can be explained in a fairly intuitive way. The intuition is that the average min-entropy evaluates the average (over all results) of the probability of the best possible guess (given the result) being correct. Here, knowing the sum, *i.e.* the number of votes for each candidate, the best guess is always the candidate with the most votes. Knowing the sum

does not provide any additional information that would help an adversary guess the first vote, compared to just knowing the winner. Therefore, the best guess regarding the first vote only depends on the winner, and the probability of it being correct is, on average, the same whether we know the sum or just the winner. The average min-entropy is thus the same in both cases.

The difference between the two cases is only in the confidence the adversary has in his guess: while, on average, the best guess is as likely to be true whether we know the sum or the winner, for a given sum, we know exactly how likely it is, while knowing the winner does not gives us this information. This means that, as reflected by the conditional Shannon entropy, the sum function does indeed leak more information than the winner function, however, as reflected by the average min-entropy, this additional information does not help an adversary make a better guess (on average) regarding the value of one specific vote.

6.8.4 Does changing votes give more power to the adversary *w.r.t.* privacy?

We finally investigate whether the adversary may gain information about the votes by changing votes, *i.e.* whether or not he has strictly more power against privacy with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$ than with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$.

We have proved that this is not the case when considering a counting function with partial tallying: we will thus consider the winner function ρ_{winner}. Following the intuition exposed earlier in this section, we study the case of the target function τ_p. That is, the adversary is trying to determine whether or not a proportion at least p of the voters who verify voted for the first candidate. Of course, if $p = 0.5$, the result immediately let the adversary know this with absolute certainty. In what follows, we will rather use $p = 0.1$.

We consider the scenario where

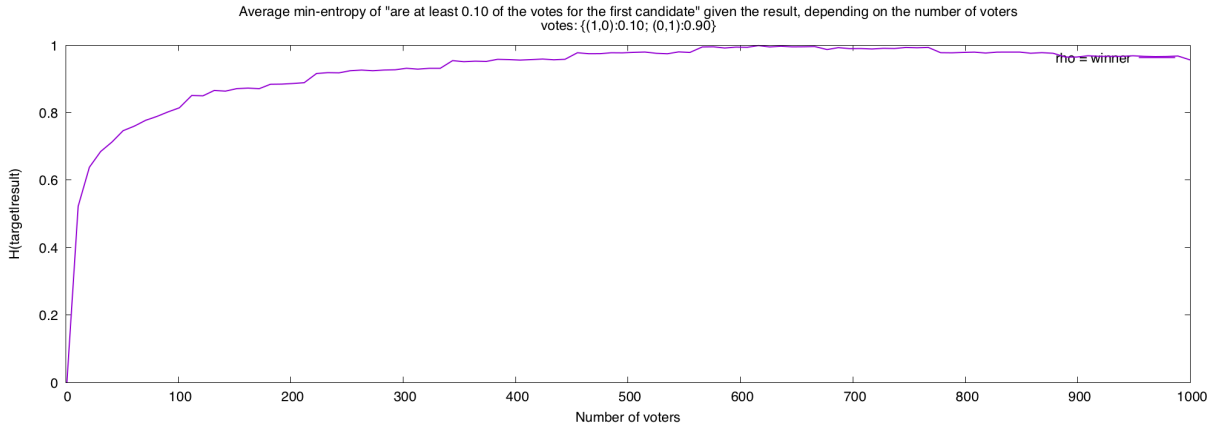
- the election involves two candidates ($d = 2$), and the initial distribution of votes (used by the environment to draw the votes at random) is 10% for the first candidate, and 90% for the second. That is, one candidate is *a priori* unpopular, and the adversary is trying to determine whether that candidate had more support among people who verify than the initial distribution would indicate.

- α voters verify, and thus their votes are unmodified;

- β voters do not check, and the adversary may change or remove their votes arbitrarily;

- there are no dishonest voters.

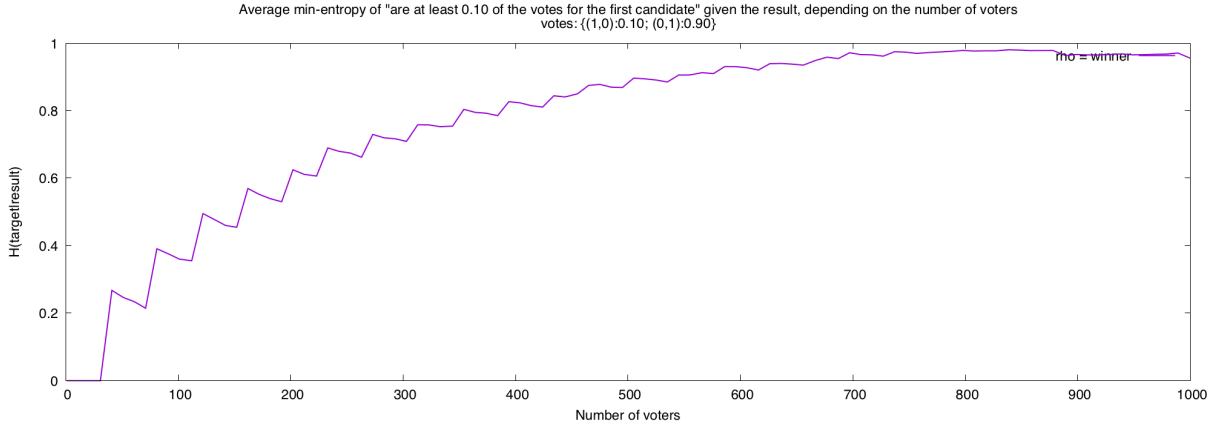
We first consider the particular case where the adversary chooses to remove all β votes, which seems to be the best he can do when interacting with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder}}$. Plotting the entropy of the target given the result, for different values of α, yields the following graph.



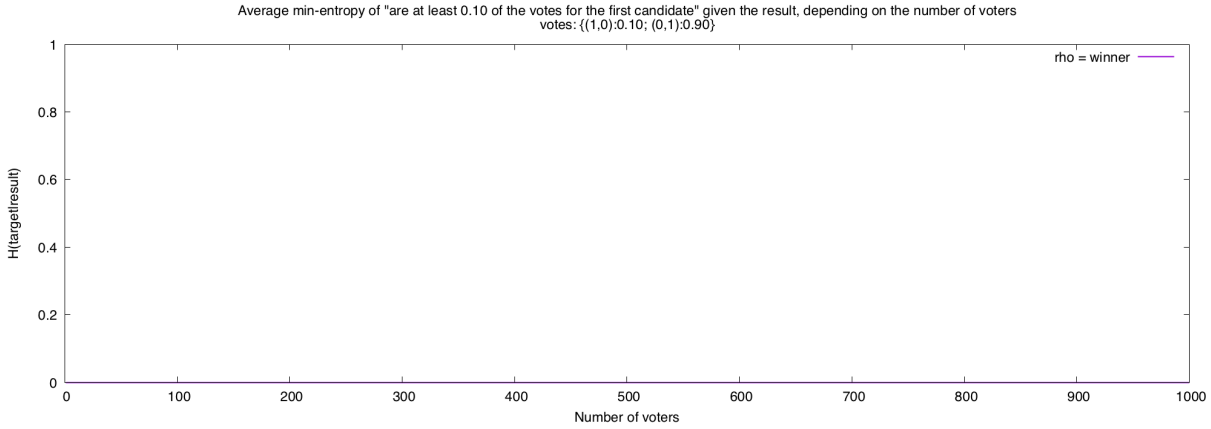
Of course, the more voters there are, the less information the result reveals about the target. We can see that the entropy seems to be slightly lower with a high number of voters: we believe this is simply an artefact of our simulation.

Now, let us consider an adversary interacting with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$, *i.e.* who can now change the value of the votes from the β voters who do not verify. Following our intuition from before, we consider the strategy that consists in turning δ of these votes into votes for the first candidate, and remove all others. Thus the result now contains α votes uncontrolled by the adversary, and δ votes for the first candidate. Here are the graphs obtained for a few values of δ (assuming β is large enough), plotted for $\alpha \in [\![0, 1000]\!]$:

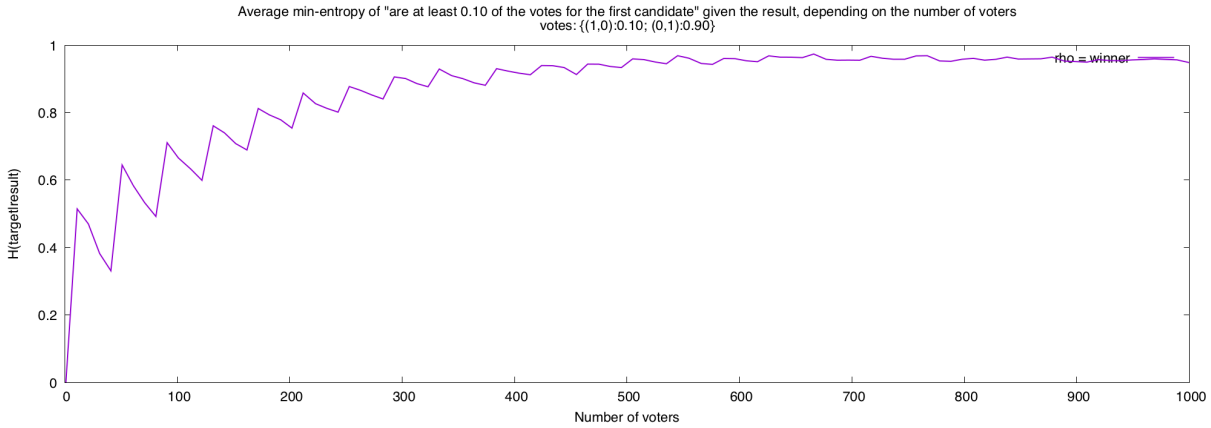
$\delta = 0.75\alpha$:



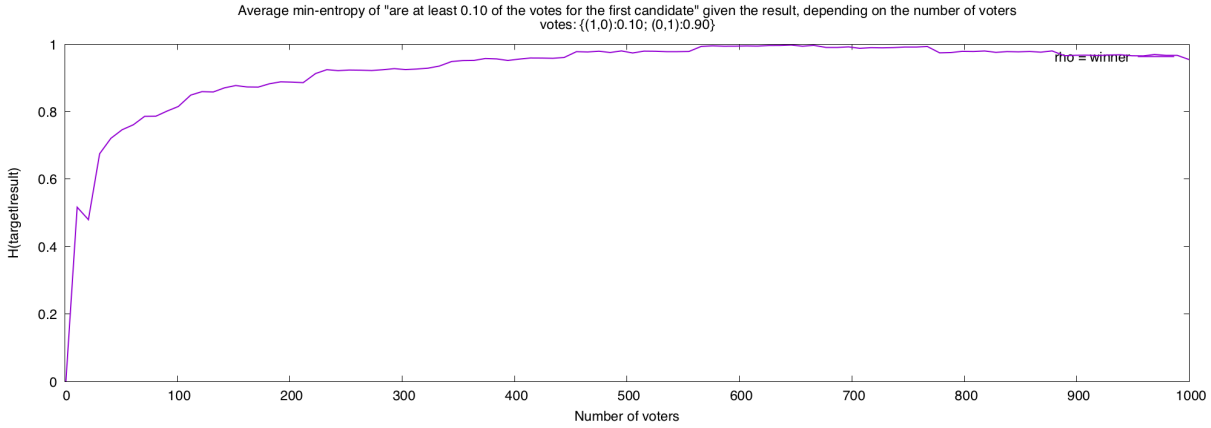
$\delta = 0.8\alpha$:



$\delta = 0.85\alpha$:



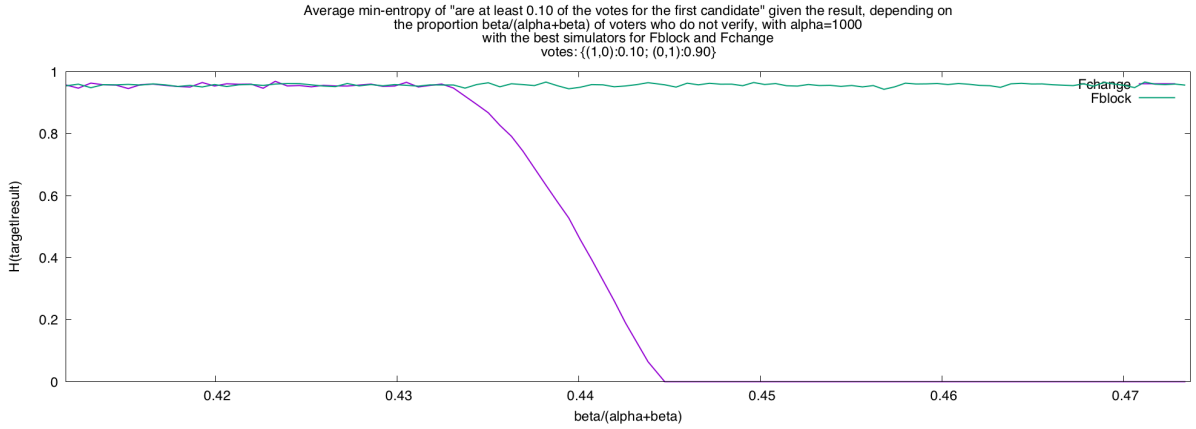
$\delta = 0.9\alpha$:



As we expected, we see that by choosing δ well, the adversary does gain more information by adding votes. By adding votes, the information the winner provides can be made closer to the target function, and the closer it is, the lower the entropy. By adding $\delta = 0.8\alpha$ votes for the first candidate, the winner gives exactly the information the adversary seeks, and the entropy is 0. Indeed if k denotes the number of honest votes (among the α) for the first candidate, the winner

lets the adversary learn whether $(k + 0.8\alpha) \geq 0.5 * (\alpha + 0.8\alpha)$, *i.e.* whether $k \geq 0.1\alpha$, which is exactly the target.

Thus, against $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$, when β is large enough, the best strategy is to turn 0.8α of the β votes into votes for the first candidate, and delete the others. Otherwise, the best strategy seems to be to turn all β votes into votes for the first candidate. On the other hand, against $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$, the most the adversary could do seems to be removing all β votes. With these strategies, for $\alpha = 1000$, plotting the average min-entropy on the target given the result as a function of the proportion $\frac{\beta}{\alpha+\beta}$ of voters who do not verify their votes, we obtain:



This shows an important difference between the two functionalities in the case of a non-partial tallying counting function, which was our goal.

In summary, these experiments with entropy show us first that, perhaps surprisingly, having the result reveal the amount of votes for each candidate may not lead to a greater loss of privacy than just revealing the winner, depending on what measure of privacy we consider (average min-entropy vs. conditional Shannon entropy). It also shows that, while we have proven before that the functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$ does not give more power to the adversary regarding privacy than $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$ in the case of counting functions with partial tallying, the same is not true (or at least, not always) when the counting function does not have this property. In the case of the winner function, provided that sufficiently many voters do not verify their votes, the adversary can learn relevant information on the votes of those who do with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$, but not with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$. To conclude this more experimental section about entropy, it appears that entropy can be a powerful tool when comparing the privacy different ideal functionality actually provide, although expressing general claims seems difficult, as this comparison largely depends on which counting function, and which target we choose to study.

Conclusion to Part II

Having studied the subtle relations between privacy and verifiability, we have proposed a flexible definition for ballot privacy against a dishonest board, that accounts for the way the adversary may try to tamper with the bulletin board. This definition describes finely the abilities of the adversary, to characterise precisely what power he has. This is not only important to analyse different schemes with various security levels but also useful when analysing a given scheme under different trust assumptions, as exemplified by our case analysis. We formally relate mb-BPRIV (with strong consistency) with a family of ideal functionalities, whose level of security is easier to understand. In the course of Chapter 6, we have considered four possible ideal functionalities, but of course other functionalities could be defined, depending on the voting scheme. Our general theorem still applies and mb-BPRIV can be proved by adapting the Recover algorithm.

Interestingly, the ideal functionalities we have proposed do not only capture vote privacy (nothing is leaked but the result) but also some form of verifiability: honest votes are deemed to be counted, at least for voters that verify. This means that mb-BPRIV can also be used to prove a certain level a verifiability. We have proved formally that these ideal functionalities, and therefore mb-BPRIV, imply a simple notion of individual verifiability against a dishonest ballot box. It would be interesting to investigate this further: which properties are missed? For instance, how does our property relate to universal verifiability? That property seems out of reach of the current definition, since we explicitly assume the tallying authority to be honest – universal verifiability aims to protect voters from dishonest talliers who attempt to tamper directly with the result itself, rather than the ballots or ballot box. A possible extension would then be to explore how to incorporate a model of dishonest talliers to our security notion. A difficulty would be that dishonest tallying authorities trivially break privacy – they know the election secret key. This makes the design of a property that could express both privacy and universal verifiability challenging. A possible approach to explore would be to design a more precise model the talliers, by representing the different trustees as separate entities, who each have only a share of the election key. Some of them would then be dishonest, *i.e.* the adversary would know their share of the key and control them, while others would remain honest, *i.e.* represented by an oracle in the game.

Another direction in which we could extend the scope of our definition is the incorporation of a distinction between humans and machines. Currently, we consider that voters are either fully dishonest – controlled by the adversary – or completely honest – *i.e.* all their computations are performed by an oracle. A scenario that is plausible in real life, but not accurately reflected in our model, is the case of a voter who is honest, but is using a compromised device to vote. Some protocols (*e.g.* BeleniosVS [56], D-Demos [51]) aim to protect the voter's privacy in such a case, *e.g.* by distributing some paper election material, containing voting codes that voters must enter on their devices to vote, rather than inputting directly their votes in clear. We could try to model such a scenario in our game by separating the oracle representing the voter from the computation that produces the ballot. For instance, the voter (oracle) could, using some

private data known only to the voter, produce some data (representing *e.g.* a voting code) that is provided to the adversary, who can use it to compute a ballot. The difficulty would then be to adapt our recovery algorithms to that setting: given a board produced by the adversary, how to determine the computation the adversary performed to obtain it from the data produced by the voters?

Finally, note that while our current definition reflects that the adversary controls the ballots cast by honest voters (to the extent permitted by the checks in place), it implicitly assumes that the voters and the auditors running the ValidBoard algorithm all have access to the *same* board, right before the tally. In practice, voters check their ballot whenever they wish to, typically right after they voted. So not only voters should always see the same board but it should grow consistently during the election. Most voting schemes assume such a setting while eluding on how to achieve this. Recent work [83, 67, 50] study this issue and propose to distribute the board. Another interesting future work would be to investigate whether guarantees ensured by mb-BPRIV for a given voting scheme are preserved when implementing it with some distributed board. To study that question, we could leverage the relation we established between mb-BPRIV and simulation-based security to build a modular proof. The first step would be to prove that a scheme, with a centralised board, is mb-BPRIV. That would imply that, with such a board, it securely implements a matching ideal functionality. That is, any execution of the scheme with the centralised board can be simulated in the ideal world. The remaining step would then be to show that the scheme *using a distributed board* securely implements the same scheme using the centralised board, *i.e.* that any execution with the distributed board can be simulated by an execution with a central board.

Conclusion

Conclusion

In this thesis we have studied different aspects of the security of remote electronic voting systems. On the one hand, the problem of automatic verification of equivalence properties, in the symbolic model. On the other hand, we also studied the definitions of vote privacy and verifiability properties, and more specifically the relations between them. Several lines of research remain to be explored to build upon the results we obtained.

1 Type system for equivalence

Part I of this thesis presents our work on the design of a type system for protocol equivalence. Our approach consists in typechecking together the two processes which we want to prove equivalent. The types we use encode information regarding the level of secrecy and integrity of messages, their structure, and even their value. Typechecking ensures the processes have the same observable behaviours. A subtlety is that this is not sufficient for them to be equivalent: the sequences of messages they produce must also be indistinguishable. This can only be enforced by having a global view of these messages, while our typing rules only apply locally to the part of the process being currently typechecked. Hence our type system collects the messages output in a *constraint*, that provides this global view. We then define a *consistency* condition to check on the constraint, that ensures the messages exchanged do not break equivalence. Our type system is sound: if two processes typecheck with consistent constraints, then they are equivalent – whether they feature bounded or unbounded numbers of sessions.

We implemented, together with Niklas Grimm, a prototype typechecker, and compared it to other tools on several examples. Our approach is much faster than most other tools in the bounded case, and on the same level of performance as ProVerif in the unbounded case, even being able to prove vote privacy for Helios, which ProVerif is unable to do, as its overapproximation leads to reporting a false attack.

Future work

As mentioned in conclusion of Part I, several extensions seem relatively accessible as short-term goals: notably, extending the process algebra and type system to handle processes with phases, or designing a type inference system, to automatically infer the types of keys.

Extension to stateful protocols Other interesting problems are more open-ended and long-term goals. Recent developments of the ProVerif tool have extended it to better deal with stateful protocols, that use a global state. Such protocols were previously out of reach of ProVerif, due to its overapproximation being to coarse to properly handle global states. Progress has been made recently in the case of trace properties, although it does not yet apply to equivalence properties. Our type system shares that limitation. It typically fails to prove equivalence of protocols with

global states, as the types do not currently allow us to express stateful information: the key types for instance simply encode information on the messages encrypted, but cannot reflect the fact that one message is encrypted before another, or that some message is only decrypted once. A possible direction to overcome this limitation would be to enrich our types to express the fact that an event is "consumed", taking ideas from linear logic.

Computational soundness Lastly, a possible line of work would be to explore how our type system could be adapted to prove computational indistinguishability, rather than symbolic equivalence. Computational soundness results for symbolic models, following the approach initially proposed by Abadi and Rogaway [9], typically consist in first relating each computational execution of the protocol to a symbolic trace, *i.e.* defining an interpretation of symbolic executions in the computational world, and showing that each computational trace is the interpretation of a symbolic trace. The second step is then roughly to show that symbolic equivalence of the messages exchanged implies the computational indistinguishability of their computational counterparts. It seems, intuitively, that our typing rules for processes impose strong conditions on the behaviours of processes, which could be sufficient, in the first step of a computational soundness proof, to ensure that the computational executions of both protocols match. The second part, regarding messages, seems more tricky. We would need to show that the typechecking of messages and consistency of the constraints entail their computational indistinguishability. Basically, our typing rules open the messages as much as an attacker possibly could, and the consistency procedure ensures the resulting subterms are cannot be distinguished. Here our approach seems somewhat similar in spirit to [9], which replaces messages with *patterns* where all parts an attacker cannot decrypt are hidden. This brings hope a similar technique could be adapted to prove the soundness of our system. Another possible approach to the problem of computational soundness would be to try to relate our system to the computationally sound symbolic attacker model from Bana and Comon. The advantage here is that rather than directly proving the computational soundness of our type system, we would show that our type system entails indistinguishability in that model, and rely on its soundness, that is already established. The idea is that this model, being symbolic, is much closer to ours than computational ones. In the Bana-Comon model, the attacker is modelled by a set of first order axioms restricting his abilities, and security is established by proving that an attacker satisfying these axioms cannot falsify a first-order formula expressing indistinguishability. A possibility to explore would be to try to relate each of our typing rules to proof steps in that logic, so that the complete typing derivation would build such a proof.

2 Privacy and individual verifiability

In Part II we have presented our work on the definitions of vote privacy and verifiability. We obtained the surprising result that privacy in fact implies individual verifiability. We proved this implication both in symbolic and computational models, to further establish that it is not a modelling artefact, but is rather a consequence of the way the privacy and verifiability properties are understood. The counter-intuitive aspect of this result may be explained by closely examining the trust assumptions. We show that privacy implies individual verifiability *with the same trust assumptions w.r.t.* the various election authorities. In contrast, verifiability is usually studied against a dishonest ballot box, while almost all existing game-based privacy definitions assume a trusted ballot box. This observation leads to our last contribution.

We proposed mb-BPRIV, a new game-based definition for vote privacy against a malicious

ballot box. Our definition is parameterised, actually forming a family of privacy notions – the parameter characterises finely which adversarial behaviours we wish not to consider as attacks. To better show that our privacy definition is still sufficiently strong and meaningful, we proved it implies a simulation-based formulation of privacy, which requires that the voting system securely implements an ideal voting functionality. We finally conducted a case study on three existing voting schemes – Helios, Belenios, and Civitas – that showcases how the flexibility of our definition lets us describe finely what guarantees are provided by each protocol.

Future work

Several interesting problems remain to be studied regarding our work on vote privacy.

Finer model of schemes and attackers First, as discussed in the conclusion of Part II, several extensions to our mb-BPRIV definition could be considered in order to cover a larger class of protocols and attackers. Notably, modelling a (partially) dishonest tallying authority, where some of the trustees are controlled by the adversary, would yield guarantees in a stronger threat model. Introducing a distinction between voters and their devices they use to vote would let us study schemes that aim to protect voters' privacy even when using a compromised voting device. Another such extension would be to tweak our property to model more finely protocols where the voting process is interactive and consists in an exchange of messages (*e.g.* the Neuchâtel protocol [78]), rather than just an algorithm running locally on the voter's machine. As mentioned in the conclusion of Part II, all of these extensions could be made by considering a more complex game, with additional oracles, yielding a finer model: separating the tallying authority into several oracles (representing each trustee); separating the voting oracle into the parts representing the voter, and the role of the device; and into several steps modelling an exchange of messages with the voting server,*etc.*

Receipt-freeness and coercion-resistance All of these extensions would yield a finer model of both the voting scheme and the adversary when expressing the same vote privacy property. Another direction to explore would be to consider different properties. As we have seen, the current mb-BPRIV already ensures more than just vote privacy: it implies that the system securely implements an ideal functionality, which has allowed us to show (under some assumptions) that it also enforces individual verifiability. An interesting line of work would be to go even further, and explore how to extend it to additionally model receipt-freeness/coercion resistance. These can be seen as stronger variants of privacy, that require that my vote remains secret even if I am forced – or willing – to give any secret data I have (credentials, passwords, *etc.*) to the adversary. In [82], Juels, Catalano and Jakobsson propose a game-based definition for coercion-resistance to study the scheme they design – JCJ, which is the basis for the Civitas protocol. Their definition however seems rather tailored to that scheme, and it is not clear that it could be used to study other protocols in general. Delaune, Kremer and Ryan [69] propose definitions for coercion-resistance and receipt-freeness in the symbolic model. These definitions are based on the one for vote privacy – secrecy of the vote is expressed as the equivalence of two scenarios where the votes are swapped. They additionally require that in the left process, Alice provides all her secret data to the attacker, while on the right process (where votes are swapped) she may provide fake data. The intuition is that the attacker should not be able to know whether real or fake data was provided, nor to use that data to learn information on the votes. An interesting future work would be to adapt this approach to our mb-BPRIV notion. Adapting the idea of voters providing real or faked data and credentials seems rather straightforward. However, an

additional subtlety arises compared to the privacy definition by swapping. Indeed, in that case, both sides of the equivalence are "symmetric" – votes are simply swapped, and thus having the real/fake credentials being provided on either side amounts to the same. In contrast, for mb-BPRIV, there *already* are a real and a fake scenario – regarding the ballots the adversary sees. There are thus two ways of writing the property: either show the adversary real ballots and data on the left, and fake them both on the right; or show him real ballots and fake data on the left, and fake ballots but real data on the right. It is not clear at all *a priori* whether these two definitions express the same property, or which one is stronger – or, in fact, whether they are incomparable or not. Further study would be required regarding this question.

Modularity All these potential extensions share a common issue though: they add more technical machinery and complexity to the definition. The mb-BPRIV game we propose is already quite complex, and extending it in such various ways would produce an even more complex definition. While not necessarily an issue when simply writing the game, the risk is that we would end up with a definition that is unusable in practice to study real protocols. What is really needed in practice is a modular definition, that is easy to adapt to many voting schemes and trust assumptions, rather than a large monolithic game that covers all possible use cases. Maybe a game-based model like ours is not the right way to proceed. An idea would be to turn more towards simulation properties – they tend to provide more modular properties, and in such models the ideal functionality makes it somewhat easier to express the guarantees they provide while abstracting from the details of the scheme. A possibility would be to design a modular simulation-based definition – with different blocks to model extensions such as those mentioned above, that would be proved separately using specific games, rather than one monolithic game as in our current definition.

Bibliography

[1] https://members.loria.fr/JLallemand/thesis/.

[2] Délibération n°2010-371 du 21 octobre 2010 portant adoption d'une recommandation relative à la sécurité des systèmes de vote électronique, 2010. French National recommendation on e-voting.

[3] Ordonnance de la ChF sur le vote électronique (OVotE) du 13 décembre 2013 (Etat le 15 janvier 2014). Chancellerie fédérale ChF, 2013. Swiss recommendation on e-voting.

[4] Délibération n° 2019-053 du 25 avril 2019 portant adoption d'une recommandation relative à la sécurité des systèmes de vote par correspondance électronique, notamment via internet, 2019. French National recommendation on e-voting.

[5] Machine readable travel document. Technical Report 9303, International Civil Aviation Organization, 2008.

[6] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM, 2001.

[7] Martín Abadi and Cédric Fournet. Private authentication. *Theoretical Computer Science*, 322(3):427 – 476, 2004.

[8] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997.*, pages 36–47, 1997.

[9] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography. In *International Conference on Theoretical Computer Science (IFIP TCS2000)*, pages 3–22. Springer, August 2000.

[10] Ben Adida. Helios: Web-based open-audit voting. In *17th USENIX Security symposium*, SS'08, pages 335–348. USENIX Association, 2008.

[11] Ben Adida, Olivier De Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. Electing a university president using open-audit voting: Analysis of real-world use of helios. In *International Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, 2009.

[12] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann.

Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*, October 2015.

[13] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Untraceability in the applied pi calculus. In *1st International Workshop on RFID Security and Cryptography*, pages 1–6. IEEE, 2009.

[14] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *2nd IEEE Computer Security Foundations Symposium (CSF'10)*, pages 107–121. IEEE Computer Society Press, 2010.

[15] Myrto Arapinis, Véronique Cortier, and Steve Kremer. When are three voters enough for privacy properties? In *21st European Symposium on Research in Computer Security (ESORICS'16)*, Lecture Notes in Computer Science, pages 241–260, Heraklion, Crete, September 2016. Springer.

[16] Myrto Arapinis, Loretta Ilaria Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New privacy issues in mobile telephony: fix and verification. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 205–216, 2012.

[17] Michael Backes, Cătălin Hritcu, and Matteo Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 195–209, Washington, DC, USA, 2008. IEEE Computer Society.

[18] David Baelde, Stéphanie Delaune, and Lucca Hirschi. Partial order reduction for security protocols. In *26th International Conference on Concurrency Theory (CONCUR'15)*, volume 42 of *LIPIcs*, pages 497–510. Leibniz-Zentrum für Informatik, 2015.

[19] Gergei Bana and Hubert Comon-Lundh. Towards unconditional soundness: Computationally complete symbolic attacker. In *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, pages 189–208, 2012.

[20] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. Probabilistic relational verification for cryptographic implementations. In *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*, pages 193–206. ACM, 2014.

[21] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 71–90, 2011.

[22] David Basin, Jannik Dreier, and Ralf Sasse. Automated Symbolic Proofs of Observational Equivalence. In *22nd ACM SIGSAC Conference on Computer and Communications Security (ACM CCS 2015)*, pages 1144–1155. ACM, October 2015.

[23] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A Formal Analysis of 5G Authentication. In *Proceedings of the 2018 ACM SIGSAC*

Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, pages 1383–1396, 2018.

[24] J. Benaloh. *Verifiable secret-ballot elections*. PhD thesis, Yale University, 1987.

[25] J. C. Benaloh and M. Yung. Distributing the power of a government to enhance the privacy of voters. In *5th ACM Symposium on Principles of Distributed Computing, (PODC'86*, pages 52—62, 1986.

[26] David Bernhard, Veronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. A comprehensive analysis of game-based ballot privacy definitions. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, pages 499–516. IEEE Computer Society Press, May 2015.

[27] David Bernhard, Véronique Cortier, Olivier Pereira, and Bogdan Warinschi. Measuring vote privacy, revisited. In *19th ACM Conference on Computer and Communications Security (CCS'12)*, pages 941–952, Raleigh, USA, October 2012.

[28] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios. In *Advances in Cryptology - ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 626–643. Springer, 2012.

[29] David Bernhard and Ben Smyth. Ballot secrecy with malicious bulletin boards. Cryptology ePrint Archive, Report 2014/822, 2014.

[30] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2014. IEEE Computer Society.

[31] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[32] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Sec. Comput.*, 5(4):193–207, 2008.

[33] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design (FOSAD'13)*, volume 8604 of *LNCS*, pages 54–87. Springer, 2013.

[34] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, 2016.

[35] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, February–March 2008.

[36] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo random bits. In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 112–117, 1982.

[37] Michele Boreale, Rocco de Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2002.

[38] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Authenticity by tagging and typing. In *2004 ACM Workshop on Formal Methods in Security Engineering*, FMSE '04, pages 1–12, New York, NY, USA, 2004. ACM.

[39] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Analysis of typed analyses of authentication protocols. In *18th IEEE Workshop on Computer Security Foundations*, CSFW '05, pages 112–125, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Sergiu Bursuc, Constantin-Cătălin Drăgan, and Steve Kremer. Private votes on untrusted platforms: models, attacks and provable scheme. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P'19)*, Stockholm, Sweden, June 2019. IEEE Computer Society Press. To appear.

[41] Ran Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, Oct 2001.

[42] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Meta-Notation for Protocol Analysis. 1 1988.

[43] Rohit Chadha, Stefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. In *Programming Languages and Systems —21th European Symposium on Programming (ESOP'12)*, volume 7211 of *Lecture Notes in Computer Science*, pages 108–127, Tallinn, Estonia, March 2012. Springer.

[44] Pyrros Chaidos, Véronique Cortier, Georg Fuchsbauer, and David Galindo. BeleniosRF: A non-interactive receipt-free electronic voting scheme. In *23rd ACM Conference on Computer and Communications Security (CCS'16)*, pages 1614–1625, Vienna, Austria, 2016.

[45] Vincent Cheval. APTE: an Algorithm for Proving Trace Equivalence. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *Lecture Notes in Computer Science*, pages 587–592, Grenoble, France, April 2014. Springer.

[46] Vincent Cheval, Véronique Cortier, and Stéphanie Delaune. Deciding equivalence-based properties using constraint solving. *Theoretical Computer Science*, 492:1–39, 2013.

[47] Vincent Cheval, Véronique Cortier, and Antoine Plet. Lengths may break privacy – or how to check for equivalences with length. In *25th International Conference on Computer Aided Verification (CAV'13)*, volume 8043 of *Lecture Notes in Computer Science*, pages 708–723, St Petersburg, Russia, July 2013. Springer.

[48] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. DEEPSEC: Deciding Equivalence Properties in Security Protocols - Theory and Practice. In *39th IEEE Symposium on Security and Privacy (S&P'18)*. IEEE Computer Society Press, 2018.

[49] Benoît Chevallier-Mames, Pierre-Alain Fouque, David Pointcheval, Julien Stern, and Jacques Traoré. On some incompatible properties of voting schemes. In *Towards Trustworthy Elections 2010*, pages 191–199, 2010.

[50] Nikos Chondros, Bingsheng Zhang, Thomas Zacharias, Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Alex Delis, Aggelos Kiayias, and Mema Roussopoulos. D-demos: A distributed, end-to-end verifiable, internet voting system. In *ICDCS 2016*, pages 711–720, 2016.

[51] Nikos Chondros, Bingsheng Zhang, Thomas Zacharias, Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Alex Delis, Aggelos Kiayias, and Mema Roussopoulos. D-DEMOS: A distributed, end-to-end verifiable, internet voting system. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016*, pages 711–720, 2016.

[52] Rémy Chrétien, Véronique Cortier, and Stéphanie Delaune. From security protocols to pushdown automata. In *Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP'13)*, volume 7966 of *Lecture Notes in Computer Science*, pages 137–149, Riga, Lituania, July 2013. Springer.

[53] John Clark and Jeremy Jacob. A survey of authentication protocol literature: Version 1.0, 1997.

[54] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy (S&P'08)*, pages 354–368. IEEE Computer Society, 2008.

[55] Véronique Cortier, Stéphanie Delaune, and Antoine Dallon. SAT-Equiv: an efficient tool for equivalence properties. In *30th IEEE Computer Security Foundations Symposium (CSF'17)*. IEEE Computer Society Press, August 2017.

[56] Véronique Cortier, Alicia Filipiak, and Joseph Lallemand. BeleniosVS: Secrecy and Verifiability against a Corrupted Voting Device. In *32nd IEEE Computer Security Foundations Symposium (CSF'19)*, Hoboken, June 2019.

[57] Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachene. Election verifiability for Helios under weaker trust assumptions. In *19th European Symposium on Research in Computer Security (ESORICS'14)*, volume 8713 of *LNCS*, pages 327–344. Springer, 2014.

[58] Véronique Cortier, David Galindo, Ralf Küsters, Johannes Müller, and Tomasz Truderung. Sok: Verifiability notions for e-voting protocols. In *36th IEEE Symposium on Security and Privacy (S&P'16)*, pages 779–798, San Jose, USA, May 2016.

[59] Véronique Cortier, Pierrick Gaudry, and Stephane Glondu. Belenios: a simple private and verifiable electronic voting system, 2019. Preprint.

[60] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A Type System for Privacy Properties. In *24th ACM Conference on Computer and Communications Security (CCS'17)*, pages 409–423. ACM, 2017.

[61] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. Equivalence properties by typing in cryptographic branching protocols. In *Proceedings of the 7th International Conference on Principles of Security and Trust (POST'18)*, pages 160–187, April 2018.

[62] Véronique Cortier, Steve Kremer, Ralf Küsters, and Bogdan Warinschi. Computationally sound symbolic secrecy in the presence of hash functions. In Naveen Garg and S. Arun-Kumar, editors, *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, volume 4337 of *Lecture Notes in Computer Science*, pages 176–187, Kolkata, India, December 2006. Springer.

[63] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3-4):225–259, April 2010.

[64] Véronique Cortier and Joseph Lallemand. Voting: You Can't Have Privacy without Individual Verifiability. In *25th ACM Conference on Computer and Communications Security (CCS'18)*, pages 53–66. ACM, 2018.

[65] Véronique Cortier and Ben Smyth. Attacking and Fixing Helios: An Analysis of Ballot Secrecy. In *24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 297–311. IEEE Computer Society Press, June 2011.

[66] Véronique Cortier and Cyrille Wiedling. A formal analysis of the norwegian e-voting protocol. In *Proceedings of the 1st International Conference on Principles of Security and Trust (POST'12)*, volume 7215 of *Lecture Notes in Computer Science*, pages 109–128. Springer, March 2012.

[67] Chris Culnane and Steve Schneider. A Peered Bulletin Board for Robust Use in Verifiable Voting Systems. In *27th IEEE Computer Security Foundations Symposium (CSF 2014)*, page 169–183, 2014.

[68] Jeremy Dawson and Alwen Tiu. Automating open bisimulation checking for the spi calculus. In *23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 307–321. IEEE Computer Society, 2010.

[69] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[70] Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 28–39, Venice, Italy, 2006. IEEE Computer Society Press.

[71] Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[72] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 463–482, 2017.

[73] Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0. RFC 2246, January 1999.

[74] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2):198–207, 1983.

[75] Nancy A. Durgin, Patrick Lincoln, John C. Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[76] Fabienne Eigner and Matteo Maffei. Differential privacy by typing in security protocols. In *26th IEEE Computer Security Foundations Symposium*, CSF '13, pages 272–286, Washington, DC, USA, 2013. IEEE Computer Society.

[77] Riccardo Focardi and Matteo Maffei. Types for security protocols. In *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, chapter 7, pages 143–181. IOS Press, 2011.

[78] David Galindo, Sandra Guasch, and Jordi Puiggali. 2015 Neuchâtel's cast-as-intended verification mechanism. In *5th International Conference on E-Voting and Identity, (VoteID'15)*, pages 3–18, 2015.

[79] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[80] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–519, July 2003.

[81] Jens Groth. Evaluating security of oting schemes in the universal composability framework. In *International Conference on Applied Cryptography and Netowrk Securtiy*, pages 46–60, 2004.

[82] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *Workshop on Privacy in the Electronic Society (WPES'05)*, pages 61–70. ACM, 2005.

[83] Aggelos Kiayias, Annabell Kuldmaa, Helger Lipmaa, Janno Siim, and Thomas Zacharias. On the security properties of e-voting bulletin boards. In *11th International Conference on Security and Cryptography for Networks (SCN 2018)*, pages 505–523, 2018.

[84] Bodo Moller, Thai Duong, Krzysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. Technical report, September 2014.

[85] Adrien Koutsos. The 5G-AKA Authentication Protocol Privacy. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 464–479, 2019.

[86] Steve Kremer, Mark D. Ryan, and Ben Smyth. Election verifiability in electronic voting protocols. In *15th European Symposium on Research in Computer Security (ESORICS'10)*, volume 6345 of *LNCS*. Springer, 2010.

[87] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Accountabiliy: Definition and relationship to verifiability. In *17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 526–535, 2010.

[88] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In *32nd IEEE Symposium on Security and Privacy (S&P 2011)*, pages 538–553. IEEE Computer Society, 2011.

[89] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, march 1996.

[90] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[91] Olivier-Pereira. *Real-World Electronic Voting: Design, Analysis and Deployment*, chapter Internet Voting with Helios. CRC Press, 2016.

[92] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[93] R. L. Rivest and W. D. Smith. Three Voting Protocols: ThreeBallot, VAV and Twin. In *USENIX/ACCURATE Electronic Voting Technology (EVT 2007)*, 2007.

[94] Peter Roenne. Private communication, 2016.

[95] Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. A Formal Definition of Protocol Indistinguishability and Its Verification Using Maude-NPA. In *STM 2014*, Lecture Notes in Computer Science, pages 162–177. IEEE Computer Society, 2014.

[96] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*, pages 256–270. ACM, 2016.

[97] Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 80–91, 1982.

Appendix

Appendix A

Proofs of the type system (Chapter 2)

In this appendix, we provide the detailed proofs to all of the theorems in Chapter 2.

Only some of the lemmas from the proof were presented in Chapter 2. The following table gives the correspondence between the numbers used for Lemmas and Theorems in Chapter 2 and the numbers used in this appendix.

Chapter 2	Appendix A
Lemma 1	Lemma A.10
Lemma 2	Lemma A.11
Lemma 3	Lemma A.22
Lemma 4	Lemma A.15
Lemma 5	Lemma A.20
Lemma 6	Lemma A.19
Lemma 7	Lemma A.21
Lemma 8	Lemma A.24

Chapter 2	Appendix A
Lemma 9	Lemma A.25
Lemma 10	Lemma A.26
Lemma 11	Theorem A.1
Theorem 1	Theorem A.2
Lemma 12	Lemma A.27
Lemma 13	Lemma A.28
Lemma 14	Theorem A.3
Lemma 15	Theorem A.4

Chapter 2	Appendix A
Theorem 2	Theorem A.5
Theorem 3	Theorem A.6
Lemma 16	Lemma A.32
Lemma 17	Lemma A.36
Lemma 18	Lemma A.39
Theorem 4	Theorem A.7
Theorem 5	Theorem A.8
Theorem 6	Theorem A.9

Unless specified otherwise, the environments Γ considered in the lemmas are implicitly assumed to be well-formed.

A.1 General results and soundness

In this section, we prove the soundness of the type system for non replicated processes, as well as several results regarding the type system that this proof uses.

Lemma A.1 (Subtyping properties)**.** *The following properties of subtyping hold:*

1. $\forall T.\quad \mathtt{HL} <: T \implies T = \mathtt{HL}$
2. $\forall T.\quad \mathtt{LL} <: T \implies T = \mathtt{LL} \vee T = \mathtt{HL}$
3. $\forall T.\quad \mathtt{HH} <: T \implies T = \mathtt{HH} \vee T = \mathtt{HL}$
4. $\forall T_1, T_2, T_3.\quad T_1 * T_2 <: T_3 \implies T_3 = \mathtt{LL} \vee T_3 = \mathtt{HL} \vee T_3 = \mathtt{HH} \vee (\exists T_4, T_5.\quad T_3 = T_4 * T_5)$
   i.e. T_3 *is* LL, HL, HH *or a pair type.*
5. $\forall T, T_1, T_2.\quad T <: T_1 * T_2 \implies (\exists T_1', T_2'.\quad T = T_1' * T_2' \wedge T_1' <: T_1 \wedge T_2' <: T_2)$

6. $\forall T_1, T_2.\quad T_1 * T_2 <: \mathtt{LL} \implies T_1 <: \mathtt{LL} \wedge T_2 <: \mathtt{LL}$

7. $\forall T_1, T_2.\quad T_1 * T_2 <: \mathtt{HH} \implies T_1 <: \mathtt{HH} \vee T_2 <: \mathtt{HH}$

8. $\forall T_1, T_2, T_3.\quad T_1 <: (T_2)_{T_3} \implies (\exists T_4 <: T_2.\quad T_1 = (T_4)_{T_3})$

9. $\forall T_1, T_2, T_3.\quad T_1 <: \{T_2\}_{T_3} \implies (\exists T_4 <: T_2.\quad T_1 = \{T_4\}_{T_3})$

10. $\forall T_1, T_2, T_3.\quad (T_1)_{T_2} <: T_3 \implies T_3 = \mathtt{HL} \vee T_3 = \mathtt{LL} \vee (\exists T_4.T_1 <: T_4 \wedge T_3 = (T_4)_{T_2})$

11. $\forall T_1, T_2, T_3.\quad \{T_1\}_{T_2} <: T_3 \implies T_3 = \mathtt{HL} \vee T_3 = \mathtt{LL} \vee (\exists T_4.T_1 <: T_4 \wedge T_3 = \{T_4\}_{T_2})$

12. $\forall T_1, T_2.\quad (T_1)_{T_2} <: \mathtt{LL} \implies T_1 <: \mathtt{LL} \wedge (T_2 = \mathtt{LL} \vee (\exists T_3.T_2 <: \mathrm{key}^{\mathtt{LL}}(T_3))$

13. $\forall T_1, T_2.\quad \{T_1\}_{T_2} <: \mathtt{LL} \implies T_1 <: \mathtt{LL} \wedge (T_2 = \mathtt{LL} \vee (\exists T_3, T_4, l.T_2 = \mathrm{pkey}(T_3) \wedge T_3 <: \mathrm{eqkey}^l(T_4))$

14. $\forall T, m, n, l, l'.\quad T <: [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!] \implies T = [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!]$

15. $\forall T, m, n, l, l'.\quad [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!] <: T \implies T = \mathtt{HL} \vee T = [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!]$

16. $\forall T_1, T_2.\quad T_1 <: T_2 \implies$ *neither* T_1 *nor* T_2 *are union types unless* $T_2 = \mathtt{HL}$ *or* $T_1 = T_2$.

17. $\forall T, l, T'.\quad T <: \mathrm{key}^l(T') \implies T = \mathrm{key}^l(T') \vee T = \mathrm{eqkey}^l(T') \vee \exists a.T = \mathrm{seskey}^{l,a}(T')$.

18. $\forall T, l, T'.\quad T <: \mathrm{eqkey}^l(T') \implies T = \mathrm{eqkey}^l(T') \vee T = \mathrm{seskey}^{l,a}(T')$.

19. $\forall T, l, a, T'.\quad T <: \mathrm{seskey}^{l,a}(T') \implies T = \mathrm{seskey}^{l,a}(T')$.

20. $\forall T, T'.\quad T <: \mathrm{pkey}(T') \implies T = \mathrm{pkey}(T')$.

21. $\forall T, T'.\quad T <: \mathrm{vkey}(T') \implies T = \mathrm{vkey}(T')$.

22. $\forall T_1, T_2, l.\quad \mathrm{key}^l(T_1) <: T_2 \implies T_2 = l \vee T_2 = \mathtt{HL} \vee T_2 = \mathrm{key}^l(T)$.

23. $\forall T_1, T_2, l.\quad \mathrm{eqkey}^l(T_1) <: T_2 \implies T_2 = l \vee T_2 = \mathtt{HL} \vee T_2 = \mathrm{key}^l(T) \vee T_2 = \mathrm{eqkey}^l(T)$.

24. $\forall T_1, T_2, l.\quad \mathrm{seskey}^{l,a}(T_1) <: T_2 \implies T_2 = l \vee T_2 = \mathtt{HL} \vee T_2 = \mathrm{key}^l(T) \vee T_2 = \mathrm{eqkey}^l(T) \vee T_2 = \mathrm{seskey}^{l,a}(T)$.

25. $\forall T_1, T_2.\quad \mathrm{pkey}(T_1) <: T_2 \implies T_2 = \mathtt{HL} \vee T_2 = \mathtt{LL} \vee T_2 = \mathrm{pkey}(T_1)$.

26. $\forall T_1.\quad \mathrm{pkey}(T_1) <: \mathtt{LL} \implies \exists T_3, l.\ T_1 <: \mathrm{eqkey}^l(T_3)$.

27. $\forall T_1, T_2.\quad \mathrm{vkey}(T_1) <: T_2 \implies T_2 = \mathtt{HL} \vee T_2 = \mathtt{LL} \vee T_2 = \mathrm{vkey}(T_1)$.

28. $\forall T_1.\quad \mathrm{vkey}(T_1) <: \mathtt{LL} \implies \exists T_3, l.\ T_1 <: \mathrm{eqkey}^l(T_3)$.

29. $\forall T.\quad T <: \mathtt{LL} \implies$ *T is a pair type* $\vee\ (\exists T', T''.\quad T = (T')_{T''}) \vee (\exists T', T''.\quad T = \{T'\}_{T''}) \vee (\exists T'.\quad T <: \mathrm{key}^{\mathtt{LL}}(T')) \vee (\exists l, T'.\quad T = \mathrm{pkey}(T')) \vee (\exists l, T'.\quad T = \mathrm{vkey}(T')) \vee T = \mathtt{LL}$.

30. $\forall T.\quad T <: \mathtt{HH} \implies T$ *is a pair type* $\vee\ (\exists T'.\quad T <: \mathrm{key}^{\mathtt{HH}}(T')) \vee T = \mathtt{HH}$.

Proof. All these properties have simple proofs by induction on the subtyping derivation. □

Lemma A.2 (Terms of type $T \vee T'$)**.** *For all* Γ, T, T', *for all ground terms* t, t', *for all* c, *if*

$$\Gamma \vdash t \sim t' : T \vee T' \to c$$

then

$$\Gamma \vdash t \sim t' : T \to c \quad \textit{or} \quad \Gamma \vdash t \sim t' : T' \to c$$

Proof. We prove this property by induction on the derivation of $\Gamma \vdash t \sim t' : T \vee T' \to c$.

The last rule of the derivation cannot be TNonce, TNonceL, TCstFN, TPubKey, TPubKeyL, TVKey, TVKeyL, TPair, TKey, TPair, TEnc, TEncH, TEncL, TAenc, TAencH, TAencL, TSignH, TSignL, THash, THashL, THigh, TLR^1, TLR$^\infty$, TLRVar, TLR', or TLRL' since the type in their conclusion cannot be $T \vee T'$. It cannot be TVar since t, t' are ground.

In the TSub case we know that $\Gamma \vdash t \sim t' : T'' \to c$ (with a shorter derivation) for some $T'' <: T \vee T'$; thus, by Lemma A.1, $T'' = T \vee T'$, and the claim holds by the induction hypothesis.

Finally in the TOr case, the premise of the rule directly proves the claim. □

Lemma A.3 (Terms and branch types)**.** *For all* Γ, T, c, *for all ground terms* t, t', *if*

$$\Gamma \vdash t \sim t' : T \to c$$

then there exists $T' \in \text{branches}(T)$ *such that*

$$\Gamma \vdash t \sim t' : T' \to c$$

Proof. This property is a corollary of Lemma A.2. We indeed prove it by successively applying this lemma to $\Gamma \vdash t \sim t' : T \to c$ until T is not a union type. □

Lemma A.4 (Substitutions type in a branch)**.** *For all* Γ, c, *for all ground substitutions* σ, σ', *if*

$$\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma_{\mathcal{X}} \to c$$

then there exists $\Gamma' \in \text{branches}(\Gamma)$ *such that*

$$\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c$$

Proof. This property follows from Lemma A.3. Indeed, by definition, $c = \bigcup_{x \in \text{dom}(\Gamma_{\mathcal{X}})} c_x$ for some c_x such that for all $x \in \text{dom}(\Gamma_{\mathcal{X}}) (= \text{dom}(\sigma) = \text{dom}(\sigma'))$,

$$\Gamma \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$$

Hence by applying Lemma A.3 we obtain a type $T_x \in \text{branches}(\Gamma(x))$ such that

$$\Gamma \vdash \sigma(x) \sim \sigma'(x) : T_x \to c_x$$

Thus if we denote Γ'' by $\forall x \in \text{dom}(\Gamma_{\mathcal{X}}).\Gamma''(x) = T_x$, and $\Gamma' = \Gamma_{\mathcal{N},\mathcal{K}} \cup \Gamma''$, we have $\Gamma' \in \text{branches}(\Gamma)$ and $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c$. □

Lemma A.5 (Typing terms in branches)**.** *For all* Γ, T, c, *for all terms* t, t', *for all* $\Gamma' \in \text{branches}(\Gamma)$, *if* $\Gamma \vdash t \sim t' : T \to c$ *then* $\Gamma' \vdash t \sim t' : T \to c$.

Corollary: in that case, there exists $T' \in \text{branches}(T)$ *such that* $\Gamma' \vdash t \sim t' : T' \to c$.

Proof. We prove this property by induction on the derivation of $\Gamma \vdash t \sim t' : T \to c$. In most cases for the last rule applied, $\Gamma(x)$ is not directly involved in the premises, for any variable x. Rather, Γ appears only in other typing judgements, or is used in $\Gamma(k, k')$ or $\Gamma(n)$ for some keys k, k' or nonce n, and keys or nonces cannot have union types. Hence, since the typing rules for terms do not change Γ, the claim directly follows from the induction hypothesis. For instance in the TPair case, we have $t = \langle t_1, t_2 \rangle$, $t' = \langle t'_1, t'_2 \rangle$, $T = T_1 * T_2$, $c = c_1 \cup c_2$, $\Gamma \vdash t_1 \sim t'_1 : T_1 \to c_1$, and $\Gamma \vdash t_2 \sim t'_2 : T_2 \to c_2$. Thus by the induction hypothesis, $\Gamma' \vdash t_1 \sim t'_1 : T_1 \to c_1$, and $\Gamma' \vdash t_2 \sim t'_2 : T_2 \to c_2$; and therefore by rule TPair, $\Gamma' \vdash t \sim t' : T \to c$. The cases of rules TPubKey, TVKey, TKey, TEnc, TEncH, TEncL, TAEnc, TAEncH, TAEncL, THashL, TSignH, TSignL, TLR', TLRL', TLRVar, TSub, TOr are similar.

The cases of rules TNonce, TNonceL, TCstFN, TPubKeyL, TVKeyL, THash, THigh, TLR^1, and TLR$^\infty$ are immediate since these rules use neither Γ nor another typing judgement in their premise.

Finally, in the TVar case, $t = t' = x$ for some variable x such that $\Gamma(x) = T$, and $c = \emptyset$. Rule TVar also proves that $\Gamma' \vdash x \sim x : \Gamma'(x) \to \emptyset$. Since $\Gamma'(x) \in \text{branches}(\Gamma(x))$, by applying rule TOr as many times as necessary, we have $\Gamma' \vdash x \sim x : \Gamma(x) \to \emptyset$, *i.e.* $\Gamma' \vdash x \sim x : T \to \emptyset$, which proves the claim.

The corollary then follows, again by induction on the typing derivation. If T is not a union type, $\text{branches}(T) = \{T\}$ and the claim is directly the previous property. Otherwise, the last rule applied in the typing derivation can only be TVar, TSub, or TOr. The TSub case follows trivially from the induction hypothesis; since T is a union type, it is its own only subtype. In the TVar case, $t = t' = x$ for some variable x such that $\Gamma(x) = T$. Hence, by definition, $\Gamma'(x) \in \text{branches}(T)$, and by rule TVar we have $\Gamma' \vdash t \sim t' : \Gamma'(x) \to c$. Finally, in the TOr case, we have $T = T_1 \vee T_2$ for some T_1, T_2 such that $\Gamma \vdash t \sim t' : T_1 \to c$. By the induction hypothesis, there exists $T'_1 \in \text{branches}(T_1)$ such that $\Gamma' \vdash t \sim t' : T'_1 \to c$. Since, by definition, $\text{branches}(T_1) \subseteq \text{branches}(T_1 \vee T_2)$, this proves the claim. □

Lemma A.6 (Typing destructors in branches)**.** *For all Γ, T, t, t', x, for all $\Gamma' \in \text{branches}(\Gamma)$, if $\Gamma \vdash t \sim t' : T$ then $\Gamma' \vdash t \sim t' : T$.*

Proof. This property is immediate by examining the typing rules for destructors. Indeed, Γ and Γ' only differ on variables, and the rules for destructors only involve $\Gamma(x)$ for $x \in \mathcal{X}$ in conditions of the form $\Gamma(x) = T$ for some type T which is not a union type.

Hence in these cases $\Gamma'(x)$ is also T, and the same rule can be applied to Γ' to prove the claim. □

Lemma A.7 (Typing processes in branches)**.** *For all Γ, C, for all processes P, Q, for all $\Gamma' \in \text{branches}(\Gamma)$, if $\Gamma \vdash P \sim Q \to C$ then there exists $C' \subseteq C$ such that $\Gamma' \vdash P \sim Q \to C'$.*

Proof. We prove this lemma by induction on the derivation of $\Gamma \vdash P \sim Q \to C$. In all the cases for the last rule applied in this derivation, we can show that the conditions of this rule still hold in Γ' (instead of Γ) using

- Lemma A.5 for the conditions of the form $\Gamma \vdash M \sim N : T \to c$;
- Lemma A.6 for the conditions of the form $\Gamma \vdash d(y) \sim d'(y) : T$;
- the fact that if $\Gamma(x)$ is not a union type, then $\Gamma'(x) = \Gamma(x)$, for conditions such as "$\Gamma(x) = \texttt{LL}$", "$\Gamma(x) = [\![\tau_m^{l,a} ; \tau_n^{l',a}]\!]$" or "$\Gamma(x) <: \text{key}^l(T)$" (in the PLetLR case);

- the induction hypothesis for the conditions of the form $\Gamma \vdash P' \sim Q' \to C''$. In this case, the induction hypothesis produces a $C''' \subseteq C''$, which can then be used to show $C' \subseteq C$, since C' and C are usually respectively C''' and C'' with some terms added.

We detail here the cases of rules POut, PPar, and POr. The other cases are similar, as explained above.

If the last rule is POut, then we have $P = \mathtt{out}(M).P'$, $Q = \mathtt{out}(N).Q'$, $C = C'' \cup_\forall c$ for some P', Q', M, N, C'', c, such that $\Gamma \vdash P' \sim Q' \to C''$ and $\Gamma \vdash M \sim N : \mathtt{LL} \to c$. Hence by Lemma A.5, $\Gamma' \vdash M \sim N : \mathtt{LL} \to c$, and by the induction hypothesis applied to P', Q', $\Gamma' \vdash P' \sim Q' \to C'''$ for some C''' such that $C''' \subseteq C''$. Therefore by rule POut, $\Gamma' \vdash P \sim Q \to C''' \cup_\forall c$, and since $C''' \cup_\forall c \subseteq C'' \cup_\forall c (= C)$, this proves the claim.

If the last rule is PPar, then we have $P = P_1 \mid P_2$, $Q = Q_1 \mid Q_2$, $C = C_1 \cup_\times C_2$ for some P_1, P_2, Q_1, Q_2, C_1, C_2 such that $\Gamma \vdash P_1 \sim Q_1 \to C_1$ and $\Gamma \vdash P_2 \sim Q_2 \to C_2$. Thus by applying the induction hypothesis twice, we have $\Gamma' \vdash P_1 \sim Q_1 \to C_1'$ and $\Gamma' \vdash P_2 \sim Q_2 \to C_2'$ with $C_1' \subseteq C_1$ and $C_2' \subseteq C_2$. Therefore by rule PPar, $\Gamma' \vdash P_1 \mid P_2 \sim Q_1 \mid Q_2 \to C_1' \cup_\times C_2'$, and since $C_1' \cup_\times C_2' \subseteq C_1 \cup_\times C_2 (= C)$, this proves the claim.

If the last rule is POr, then there exist Γ'', x, T_1, T_2, C_1 and C_2 such that $\Gamma = \Gamma'', x : T_1 \vee T_2$, $C = C_1 \cup C_2$, $\Gamma'', x : T_1 \vdash P \sim Q \to C_1$ and $\Gamma'', x : T_2 \vdash P \sim Q \to C_2$. By definition of branches, it is clear that $\mathrm{branches}(\Gamma) = \mathrm{branches}(\Gamma'', x : T_1 \vee T_2) = \mathrm{branches}(\Gamma'', x : T_1) \cup \mathrm{branches}(\Gamma'', x : T_2)$. Thus, since $\Gamma' \in \mathrm{branches}(\Gamma)$, we know that $\Gamma' \in \mathrm{branches}(\Gamma'', x : T_1)$ or $\Gamma' \in \mathrm{branches}(\Gamma'', x : T_2)$. We write the proof for the case where $\Gamma' \in \mathrm{branches}(\Gamma'', x : T_1)$, the other case is analogous. By applying the induction hypothesis to $\Gamma'', x : T_1 \vdash P \sim Q \to C_1$, there exists $C_1' \subseteq C_1$ such that $\Gamma' \vdash P \sim Q \to C_1'$. Since $C_1 \subseteq C$, this proves the claim. □

Lemma A.8 (Environments in the constraints)**.** *For all Γ, C, for all processes P, Q, if*

$$\Gamma \vdash P \sim Q \to C$$

then for all $(c, \Gamma') \in C$,

$$\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{bvars}(P) \cup \mathrm{bvars}(Q) \cup \mathrm{nnames}(P) \cup \mathrm{nnames}(Q) \cup (\mathrm{nkeys}(P) \cup \mathrm{nkeys}(Q))^2$$

(where $\mathrm{bvars}(P)$*,* $\mathrm{nnames}(P)$*,* $\mathrm{nkeys}(P)$ *respectively denote the sets of bound variables, names, and key names in P).*

Proof. We prove this lemma by induction on the typing derivation of $\Gamma \vdash P \sim Q \to C$.

If the last rule applied in this derivation is PZero, we have $C = \{(\emptyset, \Gamma)\}$, and the claim clearly holds.

In the PPar case, we have $P = P_1 \mid P_2$, $Q = Q_1 \mid Q_2$, and $C = C_1 \cup_\times C_2$ for some P_1, P_2, Q_1, Q_2, C_1, C_2 such that $\Gamma \vdash P_1 \sim Q_1 \to C_1$ and $\Gamma \vdash P_2 \sim Q_2 \to C_2$. Thus any element of C is of the form $(c_1 \cup c_2, \Gamma_1 \cup \Gamma_2)$ where $(c_1, \Gamma_1) \in C_1$, $(c_2, \Gamma_2) \in C_2$, and Γ_1, Γ_2 are compatible. By the induction hypothesis,

$$\begin{aligned}
&\mathrm{dom}(\Gamma_1) \\
&\quad \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{bvars}(P_1) \cup \mathrm{bvars}(Q_1) \cup \mathrm{nnames}(P_1) \cup \mathrm{nnames}(Q_1) \cup (\mathrm{nkeys}(P_1) \cup \mathrm{nkeys}(Q_1))^2 \\
&\quad \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{bvars}(P) \cup \mathrm{bvars}(Q) \cup \mathrm{nnames}(P) \cup \mathrm{nnames}(Q) \cup (\mathrm{nkeys}(P) \cup \mathrm{nkeys}(Q))^2,
\end{aligned}$$

and similarly for Γ_2. Therefore, since $\mathrm{dom}(\Gamma_1 \cup \Gamma_2) = \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)$ (by definition), the claim holds.

In the PIn, PLet, PLetDec, PLetAdecSame, and PLetAdecDiff cases, the typing judgement appearing in the condition of the rule uses Γ extended with an additional variable, which is bound in P and Q. We detail the PIn case, the other cases are similar. We have $P = \mathtt{in}(x).P'$, $Q = \mathtt{in}(x).Q'$ for some x, P', Q' such that $x \notin \mathrm{dom}(\Gamma)$ and $\Gamma, x : \mathtt{LL} \vdash P' \sim Q' \to C$. Hence by the induction hypothesis, if $(c, \Gamma') \in C$, then

$$\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma, x : \mathtt{LL}) \cup \mathrm{bvars}(P') \cup \mathrm{bvars}(Q') \cup \mathrm{nnames}(P') \cup \mathrm{nnames}(Q') \cup (\mathrm{nkeys}(P') \cup \mathrm{nkeys}(Q'))^2.$$

Since $\mathrm{bvars}(P) = \{x\} \cup \mathrm{bvars}(P')$ and $\mathrm{bvars}(Q) = \{x\} \cup \mathrm{bvars}(Q')$, this proves the claim.

The cases of rules PNew and PNewKey are similar, extending Γ with a nonce or key instead of a variable.

In the POut case, there exist P', Q', M, N, C', c such that $P = \mathtt{out}(M).P'$, $Q = \mathtt{out}(N).Q'$, $C = C' \cup_\forall c$, $\Gamma \vdash M \sim N : \mathtt{LL} \to c$ and $\Gamma \vdash P' \sim Q' \to C'$. If $(c', \Gamma') \in C$, by definition of $\cup_\forall$ there exists c'' such that $(c'', \Gamma') \in C'$ and $c' = c \cup c''$. By the induction hypothesis, we thus have

$$\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{bvars}(P') \cup \mathrm{bvars}(Q') \cup \mathrm{nnames}(P') \cup \mathrm{nnames}(Q')$$

and since $\mathrm{bvars}(P') = \mathrm{bvars}(P)$, $\mathrm{nnames}(P') = \mathrm{nnames}(P)$, and similarly for Q, this proves the claim.

In the PIfL case, there exist P', P'', Q', Q'', M, N, M', N', C', C'', c, c' such that $P = \mathtt{if}\ M = M'\ \mathtt{then}\ P'\ \mathtt{else}\ P''$, $Q = \mathtt{if}\ N = N'\ \mathtt{then}\ Q'\ \mathtt{else}\ Q''$, $C = (C' \cup C'') \cup_\forall (c \cup c')$, $\Gamma \vdash M \sim N : \mathtt{LL} \to c$, $\Gamma \vdash M' \sim N' : \mathtt{LL} \to c'$, $\Gamma \vdash P' \sim Q' \to C'$, and $\Gamma \vdash P'' \sim Q'' \to C''$. If $(c'', \Gamma') \in C$, by definition of $\cup_\forall$ there exist c''', such that $(c''', \Gamma') \in C' \cup C''$ and $c'' = c''' \cup c \cup c'$. We write the proof for the case where $(c''', \Gamma') \in C'$, the other case is analogous. By the induction hypothesis, we thus have

$$\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{bvars}(P') \cup \mathrm{bvars}(Q') \cup \mathrm{nnames}(P') \cup \mathrm{nnames}(Q') \cup (\mathrm{nkeys}(P') \cup \mathrm{nkeys}(Q'))^2$$

and since $\mathrm{bvars}(P') \subseteq \mathrm{bvars}(P)$, $\mathrm{nnames}(P') \subseteq \mathrm{nnames}(P)$, and similarly for Q, this proves the claim.

The cases of rules POr, PLetLR, PIfLR, PIfS, PIfLR*, PIfP, PIfI, PIfLR'*, and PIfAll remain. All these cases are similar, we write the proof for the PIfLR* case. In this case, there exist P', P'', Q', Q'', M, N, M', N', C', C'', l, l', m, n such that

- $P = \mathtt{if}\ M = M'\ \mathtt{then}\ P'\ \mathtt{else}\ P''$,
- $Q = \mathtt{if}\ N = N'\ \mathtt{then}\ Q'\ \mathtt{else}\ Q''$,
- $C = C' \cup C''$,
- $\Gamma \vdash M \sim N : [\![\tau_m^{l,\infty}\,;\, \tau_n^{l',\infty}]\!] \to \emptyset$,
- $\Gamma \vdash M' \sim N' : [\![\tau_m^{l,\infty}\,;\, \tau_n^{l',\infty}]\!] \to \emptyset$,
- $\Gamma \vdash P' \sim Q' \to C'$,
- and $\Gamma \vdash P'' \sim Q'' \to C''$.

If $(c, \Gamma') \in C$, we thus know that $(c, \Gamma') \in C'$ or $(c, \Gamma') \in C''$. We write the proof for the case where $(c, \Gamma') \in C'$, the other case is analogous. By the induction hypothesis, we thus have

$$\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{bvars}(P') \cup \mathrm{bvars}(Q') \cup \mathrm{nnames}(P') \cup \mathrm{nnames}(Q') \cup (\mathrm{nkeys}(P') \cup \mathrm{nkeys}(Q'))^2$$

and since $\mathrm{bvars}(P') \subseteq \mathrm{bvars}(P)$, $\mathrm{nnames}(P') \subseteq \mathrm{nnames}(P)$, and similarly for Q, this proves the claim. □

Lemma A.9 (Environments in the constraints do not contain union types)**.** *For all Γ, C, for all processes P, Q, if*

$$\Gamma \vdash P \sim Q \to C$$

then for all $(c, \Gamma') \in C$,

$$\mathrm{branches}(\Gamma') = \{\Gamma'\}$$

i.e. *for all $x \in \mathrm{dom}(\Gamma')$, $\Gamma'(x)$ is not a union type.*

Proof. This property is immediate by induction on the typing derivation. □

Lemma A.10 (Typing is preserved by extending the environment)**.** *For all Γ, Γ', P, Q, C, c, t, t', T, c, if $\Gamma \vdash \diamond$ and $\Gamma \cup \Gamma' \vdash \diamond$ (we do not require that Γ' is well-formed):*

- *if $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$, and if $\Gamma \vdash t \sim t' : T \to c$, then $\Gamma \cup \Gamma' \vdash t \sim t' : T \to c$.*
- *if $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$, and if $\Gamma \vdash d(y) : T$, then $\Gamma \cup \Gamma' \vdash d(y) : T$.*
- *if $(\mathrm{dom}(\Gamma) \cup \mathrm{bvars}(P) \cup \mathrm{bvars}(Q) \cup \mathrm{nnames}(P) \cup \mathrm{nnames}(Q)) \cap \mathrm{dom}(\Gamma') = \emptyset$, and $\mathrm{keys}(\Gamma') \cap (\mathrm{keys}(\Gamma) \cup \mathrm{nkeys}(P) \cup \mathrm{nkeys}(Q)) = \emptyset$, and $\Gamma \vdash P \sim Q \to C$, then $\Gamma \cup \Gamma' \vdash P \sim Q \to C'$. where $C' = \{(c, \Gamma_c \cup \Gamma'') | (c, \Gamma_c) \in C \wedge \Gamma'' \in \mathrm{branches}(\Gamma')\}$ (note that the union is well defined,* i.e. *Γ_c and Γ'' are compatible, thanks to Lemma A.8)*

Proof.

- The first point is immediate by induction on the type derivation.
- The second point is immediate by examining the typing rules for destructors.
- The third point is immediate by induction on the type derivation of the processes. In the PZero case, to satisfy the condition that the environment is its own only branch, rule POr needs to be applied first, in order to split all the union types in Γ', which yields the environments $\mathrm{branches}(\Gamma \cup \Gamma')$ in the constraints.

□

Lemma A.11 (Consistency for Subsets)**.** *The following statements about constraints hold:*

1. *If (c, Γ) is consistent, and $c' \subseteq c$ then (c', Γ) is consistent.*
2. *Let C be a consistent constraint set. Then every subset $C' \subseteq C$ is also consistent.*
3. *If $C \cup_\forall c'$ is consistent then C also is.*
4. *If $C_1 \subseteq C_2$ and $C_1' \subseteq C_2'$, then $C_1 \cup_\times C_1' \subseteq C_2 \cup_\times C_2'$.*
5. *$[\![\cdot]\!]_{\sigma,\sigma'}$ commutes with $\cup$, $\cup_\times$, $\cup_\forall$,* i.e. *for all C, C', σ, σ', $[\![C \cup_\times C']\!]_{\sigma,\sigma'} = [\![C]\!]_{\sigma,\sigma'} \cup_\times [\![C']\!]_{\sigma,\sigma'}$ and similarly for $\cup$, $\cup_\forall$.*

6. *If σ_1 and σ'_1 are ground and have disjoint domains, as well as σ_2 and σ'_2, then for all c,* $\left[\!\left[[\![c]\!]_{\sigma_1,\sigma_2} \right]\!\right]_{\sigma'_1,\sigma'_2} = [\![c]\!]_{\sigma_1\cup\sigma'_1,\sigma_2\cup\sigma'_2}$

7. *if σ, σ' are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma_{\mathcal{X}} \to c$ for some c, if $C \cup_\forall c$ is consistent, and if for all $(c', \Gamma') \in C$, $\Gamma \subseteq \Gamma'$, then $[\![C]\!]_{\sigma,\sigma'}$ is consistent.*

Proof. Points 1 and 2 follow immediately from the definition of consistency and of static equivalence.

Point 3 follows from point 1: for every $(c, \Gamma) \in C$, $(c \cup c', \Gamma)$ is in $C \cup_\forall c'$, and is therefore consistent. Hence (c, Γ) also is by point 1.

Point 4 follows from the definition of $\cup_\times$. If $(c, \Gamma) \in C_1 \cup_\times C'_1$, there exists $(c_1, \Gamma_1) \in C_1$, $(c'_1, \Gamma'_1) \in C'_1$ such that $(c, \Gamma) = (c_1 \cup c'_1, \Gamma_1 \cup \Gamma'_1)$ (and Γ_1, Γ'_1 are compatible). Since $C_1 \subseteq C_2$, $(c_1, \Gamma_1) \in C_2$. Similarly, $(c'_1, \Gamma'_1) \in C'_2$. Therefore $(c, \Gamma) \in C_2 \cup_\times C'_2$.

Points 5 and 6 follow from the definitions of $[\![\cdot]\!]_{\sigma,\sigma'}$, $\cup_\times$, $\cup_\forall$.

Point 7 follows from the definitions of $[\![\cdot]\!]_{\sigma,\sigma'}$, and of consistency. Indeed, let $(c', \Gamma') \in [\![C]\!]_{\sigma,\sigma'}$. There exists c'' such that $c' = [\![c'']\!]_{\sigma,\sigma'}$, and $(c'', \Gamma') \in C$. Let $c_1 \subseteq c'$, and $\Gamma_1 \subseteq \Gamma'$ such that $\Gamma_{1\mathcal{N},\mathcal{K}} = \Gamma'_{\mathcal{N},\mathcal{K}}$ and $\mathrm{vars}(c_1) \subseteq \mathrm{dom}(\Gamma_1)$. Let θ, θ', c_θ be such that $(\Gamma_1)_{\mathcal{N},\mathcal{K}} \vdash \theta \sim \theta' : (\Gamma_1)_{\mathcal{X}} \to c_\theta$, and $c_\theta \subseteq [\![c_1]\!]_{\theta,\theta'}$.

Note that since σ, σ' are ground, c is also ground.

Since $c' = [\![c'']\!]_{\sigma,\sigma'}$, there exists $c_2 \subseteq c''$ such that $c_1 = [\![c_2]\!]_{\sigma,\sigma'}$. If we show that there exists c_3 such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma\theta \sim \sigma'\theta' : \Gamma_{\mathcal{X}} \to c_3$ and $c_3 \subseteq [\![c_2 \cup c]\!]_{\sigma\theta,\sigma'\theta'}$, then it will follow from the consistency of $C \cup_\forall c$ that $\phi_{\mathrm{LL}}^{\Gamma_2} \cup \phi_\ell(c_2 \cup c)\sigma\theta$ and $\phi_{\mathrm{LL}}^{\Gamma_2} \cup \phi_r(c_2 \cup c)\sigma'\theta'$ are statically equivalent, where $\Gamma_2 = \Gamma_1 \cup \Gamma \subseteq \Gamma''$.

Since $\Gamma_{1\mathcal{N},\mathcal{K}} = \Gamma''_{\mathcal{N},\mathcal{K}}$ and $\Gamma \subseteq \Gamma''$, we have $\phi_{\mathrm{LL}}^{\Gamma_2} = \phi_{\mathrm{LL}}^{\Gamma_1}$. Hence $\phi_{\mathrm{LL}}^{\Gamma_1} \cup \phi_\ell(c_1 \cup c)\theta$ and $\phi_{\mathrm{LL}}^{\Gamma_1} \cup \phi_r(c_1 \cup c)\theta'$ will be statically equivalent.

Therefore, $\phi_{\mathrm{LL}}^{\Gamma_1} \cup \phi_\ell(c_1)\theta$ and $\phi_{\mathrm{LL}}^{\Gamma_1} \cup \phi_r(c_1)\theta'$ will also be statically equivalent, which proves the consistency of $[\![C]\!]_{\sigma,\sigma'}$.

It only remains to be proved that there exists c_3 such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma\theta \sim \sigma'\theta' : \Gamma_{\mathcal{X}} \to c_3$ and $c_3 \subseteq [\![c_2 \cup c]\!]_{\sigma\theta,\sigma'\theta'}$.

Since σ is ground, $\sigma\theta = \sigma \cup \theta|_{\mathrm{dom}(\Gamma_1)\setminus\mathrm{dom}(\Gamma)}$, and similarly for $\sigma'\theta'$. By assumption, $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma_{\mathcal{X}} \to c$ and $(\Gamma_1)_{\mathcal{N},\mathcal{K}} \vdash \theta \sim \theta' : (\Gamma_1)_{\mathcal{X}} \to c_\theta$. Hence there exists $c_3 \subseteq c \cup c_\theta$ such that $(\Gamma_2)_{\mathcal{N},\mathcal{K}} \vdash \sigma\theta \sim \sigma'\theta' : (\Gamma_2)_{\mathcal{X}} \to c_3$.

Since $c_\theta \subseteq [\![c_1]\!]_{\theta,\theta'}$, and c is ground, we have

$$\begin{aligned} c_3 &\subseteq c \cup [\![c_1]\!]_{\theta,\theta'} \\ &= c \cup [\![c_2]\!]_{\sigma\theta,\sigma'\theta'} \\ &= [\![c_2 \cup c]\!]_{\sigma\theta,\sigma'\theta'} \end{aligned}$$

which concludes the proof. □

Lemma A.12 (Environments in constraints contain a branch of the typing environment)**.** *For all Γ, C, for all processes P, Q, if $\Gamma \vdash P \sim Q \to C$ then for all $(c, \Gamma') \in C$, there exists $\Gamma'' \in \mathrm{branches}(\Gamma)$ such that $\Gamma'' \subseteq \Gamma'$.*

Proof. We prove this property by induction on the type derivation of $\Gamma \vdash P \sim Q \to C$. In the PZero case, $C = \{(\emptyset, \Gamma)\}$, and by assumption $\mathrm{branches}(\Gamma) = \{\Gamma\}$, hence the claim trivially holds.

In the PPar case, we have $P = P_1 \mid P_2$, $Q = Q_1 \mid Q_2$, and $C = C_1 \cup_\times C_2$ for some P_1, P_2, Q_1, Q_2, C_1, C_2 such that $\Gamma \vdash P_1 \sim Q_1 \to C_1$ and $\Gamma \vdash P_2 \sim Q_2 \to C_2$. Thus any element of C is of the form $(c_1 \cup c_2, \Gamma_1 \cup \Gamma_2)$ where $(c_1, \Gamma_1) \in C_1$, $(c_2, \Gamma_2) \in C_2$, and Γ_1, Γ_2 are compatible. By the induction hypothesis, both C_1 and C_2 contain a branch of Γ. The claim holds, as these are necessarily the same branch, since Γ_1 and Γ_2 are compatible.

In the POr case, we have $\Gamma = \Gamma'', x : T_1 \vee T_2$ for some x, Γ'', T_1, T_2 such that $\Gamma'', x : T_1 \vdash P \sim Q \to C_1$ and $\Gamma'', x : T_2 \vdash P \sim Q \to C_2$, and $C = C_1 \cup C_2$. Thus by the induction hypothesis, if $(c, \Gamma') \in C_i$ (for $i \in \{1, 2\}$), then Γ' contains some $\Gamma''' \in \text{branches}(\Gamma'', x : T_i) \subseteq \text{branches}(\Gamma)$, and the claim holds.

In the POut case, there exist P', Q', M, N, C', c such that $P = \texttt{out}(M).P'$, $Q = \texttt{out}(N).Q'$, $C = C' \cup_\forall c$, $\Gamma \vdash M \sim N : \texttt{LL} \to c$ and $\Gamma \vdash P' \sim Q' \to C'$. If $(c', \Gamma') \in C$, by definition of $\cup_\forall$ there exists c'' such that $(c'', \Gamma') \in C'$ and $c' = c \cup c''$. Hence by applying the induction hypothesis to $\Gamma \vdash P' \sim Q' \to C'$, there exists $\Gamma'' \in \text{branches}(\Gamma)$ such that $\Gamma'' \subseteq \Gamma'$.

In the PIfL case, there exist P', P'', Q', Q'', M, N, M', N', C', C'', c, c' such that $P = \texttt{if } M = M' \texttt{ then } P' \texttt{ else } P''$, $Q = \texttt{if } N = N' \texttt{ then } Q' \texttt{ else } Q''$, $C = (C' \cup C'') \cup_\forall (c \cup c')$, $\Gamma \vdash M \sim N : \texttt{LL} \to c$, $\Gamma \vdash M' \sim N' : \texttt{LL} \to c'$, $\Gamma \vdash P' \sim Q' \to C'$, and $\Gamma \vdash P'' \sim Q'' \to C''$. If $(c'', \Gamma') \in C$, by definition of $\cup_\forall$ there exist c''', such that $(c''', \Gamma') \in C' \cup C''$ and $c'' = c''' \cup c \cup c'$. We write the proof for the case where $(c''', \Gamma') \in C'$, the other case is analogous. By applying the induction hypothesis to $\Gamma \vdash P' \sim Q' \to C'$, there exists $\Gamma'' \in \text{branches}(\Gamma)$ such that $\Gamma'' \subseteq \Gamma'$, which proves the claim.

All remaining cases are similar. We write the proof for the PIfLR* case. In this case, there exist P', P'', Q', Q'', M, N, M', N', C', C'', l, l', m, n such that $P = \texttt{if } M = M' \texttt{ then } P' \texttt{ else } P''$, $Q = \texttt{if } N = N' \texttt{ then } Q' \texttt{ else } Q''$, $C = C' \cup C''$, $\Gamma \vdash M \sim N : [\![\tau_m^{l,\infty} ; \tau_n^{l',\infty}]\!] \to \emptyset$, $\Gamma \vdash M' \sim N' : [\![\tau_m^{l,\infty} ; \tau_n^{l',\infty}]\!] \to \emptyset$, $\Gamma \vdash P' \sim Q' \to C'$, and $\Gamma \vdash P'' \sim Q'' \to C''$. If $(c, \Gamma') \in C$, we thus know that $(c, \Gamma') \in C'$ or $(c, \Gamma') \in C''$. We write the proof for the case where $(c, \Gamma') \in C'$, the other case is analogous. By applying the induction hypothesis to $\Gamma \vdash P' \sim Q' \to C'$, there exists $\Gamma'' \in \text{branches}(\Gamma)$ such that $\Gamma'' \subseteq \Gamma'$, which proves the claim. □

Lemma A.13 (All branches are represented in the constraints)**.** *For all Γ, C, for all processes P, Q, if $\Gamma \vdash P \sim Q \to C$ then for all $\Gamma' \in \text{branches}(\Gamma)$, there exists $(c, \Gamma'') \in C$, such that $\Gamma' \subseteq \Gamma''$.*

Proof. We prove this property by induction on the type derivation of $\Gamma \vdash P \sim Q \to C$. In the PZero case, $C = \{(\emptyset, \Gamma)\}$, and by assumption $\text{branches}(\Gamma) = \{\Gamma\}$, hence the claim trivially holds.

In the PPar case, we have $P = P_1 \mid P_2$, $Q = Q_1 \mid Q_2$, and $C = C_1 \cup_\times C_2$ for some P_1, P_2, Q_1, Q_2, C_1, C_2 such that $\Gamma \vdash P_1 \sim Q_1 \to C_1$ and $\Gamma \vdash P_2 \sim Q_2 \to C_2$. By the induction hypothesis, there exists $(c_1, \Gamma_1) \in C_1$ and $(c_2, \Gamma_2) \in C_2$ such that $\Gamma' \subseteq \Gamma_1$ and $\Gamma' \subseteq \Gamma_2$. By Lemma A.8, $\text{dom}(\Gamma_1)$ and $\text{dom}(\Gamma_2)$ only contain $\text{dom}(\Gamma) (= \text{dom}(\Gamma'))$ and variables and names in

$$\text{bvars}(P_1) \cup \text{bvars}(Q_1) \cup \text{nnames}(P_1) \cup \text{nnames}(Q_1) \cup (\text{nkeys}(P_1) \cup \text{nkeys}(Q_1))^2$$

and

$$\text{bvars}(P_2) \cup \text{bvars}(Q_2) \cup \text{nnames}(P_2) \cup \text{nnames}(Q_2) \cup (\text{nkeys}(P_2) \cup \text{nkeys}(Q_2))^2$$

respectively. We have $\Gamma_1(x) = \Gamma_2(x) = \Gamma'(x)$ for all $x \in \text{dom}(\Gamma')$, and the sets

$$\text{bvars}(P_1) \cup \text{bvars}(Q_1) \cup \text{nnames}(P_1) \cup \text{nnames}(Q_1) \cup (\text{nkeys}(P_1) \cup \text{nkeys}(Q_1))^2$$

and

$$\mathrm{bvars}(P_2) \cup \mathrm{bvars}(Q_2) \cup \mathrm{nnames}(P_2) \cup \mathrm{nnames}(Q_2) \cup (\mathrm{nkeys}(P_2) \cup \mathrm{nkeys}(Q_2))^2$$

are disjoint by well formedness of the processes $P_1 \mid P_2$ and $Q_1 \mid Q_2$. Thus Γ_1 and Γ_2 are compatible. Therefore $(c_1 \cup c_2, \Gamma_1 \cup \Gamma_2) \in C_1 \cup_\times C_2 (= C)$, and the claim holds since $\Gamma' \subseteq \Gamma_1 \cup \Gamma_2$.

In the POR case, we have $\Gamma = \Gamma'', x : T_1 \vee T_2$ for some x, Γ'', T_1, T_2 such that $\Gamma'', x : T_1 \vdash P \sim Q \to C_1$ and $\Gamma'', x : T_2 \vdash P \sim Q \to C_2$, and $C = C_1 \cup C_2$. Since $\mathrm{branches}(\Gamma) = \mathrm{branches}(\Gamma'', x : T_1) \cup \mathrm{branches}(\Gamma'', x : T_2)$, we know that $\Gamma' \in \mathrm{branches}(\Gamma'', x : T_i)$ for some i. We conclude this case directly by applying the induction hypothesis to $\Gamma'', x : T_i \vdash P \sim Q \to C_i$.

In the POUT case, there exist P', Q', M, N, C', c such that $P = \mathtt{out}(M).P'$, $Q = \mathtt{out}(N).Q'$, $C = C' \cup_\forall c$, $\Gamma \vdash M \sim N : \mathtt{LL} \to c$ and $\Gamma \vdash P' \sim Q' \to C'$. By applying the induction hypothesis to $\Gamma \vdash P' \sim Q' \to C'$, there exists $(c'', \Gamma'') \in C'$ such that $\Gamma' \subseteq \Gamma''$. By definition of $\cup_\forall$, $(c'' \cup c, \Gamma'') \in C$, which proves the claim.

In the PIFL case, there exist P', P'', Q', Q'', M, N, M', N', C', C'', c, c' such that $P = \mathtt{if}\ M = M'\ \mathtt{then}\ P'\ \mathtt{else}\ P''$, $Q = \mathtt{if}\ N = N'\ \mathtt{then}\ Q'\ \mathtt{else}\ Q''$, $C = (C' \cup C'') \cup_\forall (c \cup c')$, $\Gamma \vdash M \sim N : \mathtt{LL} \to c$, $\Gamma \vdash M' \sim N' : \mathtt{LL} \to c'$, $\Gamma \vdash P' \sim Q' \to C'$, and $\Gamma \vdash P'' \sim Q'' \to C''$. By applying the induction hypothesis to $\Gamma \vdash P' \sim Q' \to C'$, there exists $(c'', \Gamma'') \in C'$ such that $\Gamma' \subseteq \Gamma''$. By definition of $\cup_\forall$, $(c'' \cup c \cup c', \Gamma'') \in C$, which proves the claim.

All remaining cases are similar. We write the proof for the PIFLR* case. In this case, there exist P', P'', Q', Q'', M, N, M', N', C', C'', l, l', m, n such that $P = \mathtt{if}\ M = M'\ \mathtt{then}\ P'\ \mathtt{else}\ P''$, $Q = \mathtt{if}\ N = N'\ \mathtt{then}\ Q'\ \mathtt{else}\ Q''$, $C = C' \cup C''$, $\Gamma \vdash M \sim N : [\![\tau_m^{l,\infty} ; \tau_n^{l',\infty}]\!] \to \emptyset$, $\Gamma \vdash M' \sim N' : [\![\tau_m^{l,\infty} ; \tau_n^{l',\infty}]\!] \to \emptyset$, $\Gamma \vdash P' \sim Q' \to C'$, and $\Gamma \vdash P'' \sim Q'' \to C''$. By applying the induction hypothesis to $\Gamma \vdash P' \sim Q' \to C'$, there exists $(c'', \Gamma'') \in C'$ such that $\Gamma' \subseteq \Gamma''$.

If $(c, \Gamma') \in C$, we thus know that $(c, \Gamma') \in C'$ or $(c, \Gamma') \in C''$. We write the proof for the case where $(c, \Gamma') \in C'$, the other case is analogous. By applying the induction hypothesis to $\Gamma \vdash P' \sim Q' \to C'$, there exists $\Gamma'' \in \mathrm{branches}(\Gamma)$ such that $\Gamma'' \subseteq \Gamma'$, which proves the claim. □

Lemma A.14 (Refinement types)**.** *For all Γ, for all terms t, t', for all m, n, a, l, l', c, if $\Gamma \vdash t \sim t' : [\![\tau_m^{l,a} ; \tau_n^{l',a}]\!] \to c$ then $c = \emptyset$ and*

- *either $t = m$, $t' = n$, $a = \infty$ and $\Gamma(m) = \tau_m^{l,a}$ and $\Gamma(n) = \tau_n^{l',a}$;*
- *or $t = m$, $t' = n$, $a = 1$, and $(\Gamma(m) = \tau_m^{l,a}) \vee (m \in \mathcal{FN} \cup \mathcal{C} \wedge l = \mathtt{LL})$, and $(\Gamma(n) = \tau_n^{l',a}) \vee (n \in \mathcal{FN} \cup \mathcal{C} \wedge l' = \mathtt{LL})$;*
- *or t and t' are variables $x, y \in \mathcal{X}$ and there exist labels l'', l''', and names m', n' such that $\Gamma(x) = [\![\tau_m^{l,a} ; \tau_{n'}^{l'',a}]\!]$ and $\Gamma(y) = [\![\tau_{m'}^{l''',a} ; \tau_n^{l',a}]\!]$.*

In particular if t, t' are ground then the last case cannot occur.

Proof. The proof of this property is immediate by induction on the typing derivation for the terms. Indeed, because of the form of the type, and by well-formedness of Γ, the only rules which can lead to $\Gamma \vdash t \sim t' : [\![\tau_m^{l,a} ; \tau_n^{l',a}]\!] \to c$ are TVAR, TLR^1, TLR$^\infty$, TLRVAR, and TSUB.

In the TVAR, TLR^1, TLR$^\infty$ cases the claim directly follows from the premises of the rule.

In the TLRVAR case, t and t' are necessarily variables, and their types in Γ are obtained directly by applying the induction hypothesis to the premises of the rule.

Finally in the TSUB case, $\Gamma \vdash t \sim t' : T \to c$ and $T <: [\![\tau_m^{l,a} ; \tau_n^{l',a}]\!]$. By Lemma A.1, $T = [\![\tau_m^{l,a} ; \tau_n^{l',a}]\!]$ and we conclude by the induction hypothesis. □

Lemma A.15 (Encryption types)**.** *For all environment* Γ*, types* T*,* T'*, messages* M*,* N*,* M_1*,* M_2 *and set of constraints* c*:*

1. *If* $\Gamma \vdash M \sim N : (T)_{T'} \to c$ *then*
   - *either there exist* M_1*,* M_2*,* N_1*,* N_2*,* c_1*,* c_2 *such that* $M = \mathtt{enc}(M_1, M_2)$*,* $N = \mathtt{enc}(N_1, N_2)$*,* $c = c_1 \cup c_2$*,* $\Gamma \vdash M_1 \sim N_1 : T \to c_1$*, and* $\Gamma \vdash M_2 \sim N_2 : T' \to c_2$*, both with shorter derivations (than the one for* $\Gamma \vdash M \sim N : (T)_{T'} \to c$*);*
   - *or* M *and* N *are variables.*
2. *If* $\Gamma \vdash M \sim N : \{T\}_{T'} \to c$ *then*
   - *either there exist* M_1*,* M_2*,* N_1*,* N_2*,* c_1*,* c_2 *such that* $M = \mathtt{aenc}(M_1, M_2)$*,* $N = \mathtt{aenc}(N_1, N_2)$*,* $c = c_1 \cup c_2$*,* $\Gamma \vdash M_1 \sim N_1 : T \to c_1$ *and* $\Gamma \vdash M_2 \sim N_2 : T' \to c_2$*, both with shorter derivations (than the one for* $\Gamma \vdash M \sim N : \{T\}_{T'} \to c$*);*
   - *or* M *and* N *are variables.*
3. *If* $T <: \mathtt{LL}$ *and* $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : T \to c$ *then* $T = \mathtt{LL}$*.*
4. *If* $T <: \mathtt{LL}$ *and* $\Gamma \vdash \mathtt{aenc}(M_1, M_2) \sim N : T \to c$ *then* $T = \mathtt{LL}$*.*
5. *If* $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : \mathtt{LL} \to c$ *then there exist* N_1*,* N_2 *such that* $N = \mathtt{enc}(N_1, N_2)$*, and*
   - *either there exist* T'*,* c_1*,* c_2 *such that* $\Gamma \vdash M_2 \sim N_2 : \mathrm{key}^{\mathtt{HH}}(T') \to c_2$*,* $c = \{\mathtt{enc}(M_1, M_2) \sim N\} \cup c_1 \cup c_2$*, and* $\Gamma \vdash M_1 \sim N_1 : T' \to c_1$*;*
   - *or there exist* c_1*,* c_2 *such that* $c = c_1 \cup c_2$*,* $\Gamma \vdash M_2 \sim N_2 : \mathtt{LL} \to c_2$ *and* $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$*.*
6. *If* $\Gamma \vdash \mathtt{aenc}(M_1, M_2) \sim N : \mathtt{LL} \to c$ *then there exist* N_1*,* N_2 *such that* $N = \mathtt{aenc}(N_1, N_2)$*, and*
   - *either there exist* T'*,* T''*,* c_1*,* c_2 *such that* $T' <: \mathrm{key}^{\mathtt{HH}}(T'')$*,* $\Gamma \vdash M_2 \sim N_2 : \mathrm{pkey}(T') \to c_2$*,* $c = \{\mathtt{aenc}(M_1, M_2) \sim N\} \cup c_1 \cup c_2$*, and* $\Gamma \vdash M_1 \sim N_1 : T'' \to c_1$*;*
   - *or there exist* c_1*,* c_2 *such that* $c = c_1 \cup c_2$*,* $\Gamma \vdash M_2 \sim N_2 : \mathtt{LL} \to c_2$*, and* $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$*.*
7. *The symmetric properties to the previous four points,* i.e. *when the term on the right is an encryption, also hold.*

Proof. We prove point 1 by induction on the derivation of $\Gamma \vdash M \sim N : (T)_{T'} \to c$. Because of the form of the type, and by well-formedness of Γ, the only possibilities for the last rule applied are TVar, TEnc, and TSub. The claim clearly holds in the TVar and TEnc cases. In the TSub case, we have $\Gamma \vdash M \sim N : T'' \to c$ for some $T'' <: (T)_{T'}$. Hence by Lemma A.1, there exists $T''' <: T$ such that $T'' = (T''')_{T'}$. Therefore, by applying the induction hypothesis to $\Gamma \vdash M \sim N : T'' \to c$

- either M and N are two variables, and the claim holds;
- or there exist M_1, M_2, N_1, N_2, c_1, c_2 such that $M = \mathtt{enc}(M_1, M_2)$, $N = \mathtt{enc}(N_1, N_2)$, $c = c_1 \cup c_2$, $\Gamma \vdash M_1 \sim N_1 : T''' \to c_1$, and $\Gamma \vdash M_2 \sim N_2 : T' \to c_2$, both with shorter derivations than the one for $\Gamma \vdash M \sim N : T'' \to c$. Thus by subtyping (rule TSub), $\Gamma \vdash M_1 \sim N_1 : T \to c_1$ with a shorter derivation that $\Gamma \vdash M \sim N : (T)_{T'} \to c$, which proves the property.

Point 2 has a similar proof to point 1.

We now prove point 3 by induction on the proof of $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : T \to c$. Because of the form of the terms, the last rule applied can only be THIGH, TOR, TENC, TENCH, TENCL, TAENCH, TAENCL, TLR', TLRL' or TSUB.

The THIGH, TLR', TOR, TENC cases are actually impossible by Lemma A.1, since $T <: \mathtt{LL}$. In the TSUB case, we have $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : T' \to c$ for some T' such that $T' <: T$. By transitivity of $<:$, $T' <: \mathtt{LL}$, and the induction hypothesis proves the claim. In all other cases, $T = \mathtt{LL}$ and the claim holds.

Point 4 has a similar proof to point 3.

We prove point 5 by induction on the proof of $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : \mathtt{LL} \to c$. Because of the form of the terms and of the type (*i.e.* LL) the last rule applied can only be TENCH, TENCL, TAENCH, TAENCL, TLRL' or TSUB.

The TLRL' case is impossible, since by Lemma A.14 it would imply that $\mathtt{enc}(M_1, M_2)$ is either a variable or a nonce.

In the TSUB case, we have $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : T' \to c$ for some T' such that $T' <: \mathtt{LL}$. By point 3, $T' = \mathtt{LL}$, and the premise of the rule thus gives a shorter derivation of $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : \mathtt{LL} \to c$. The induction hypothesis applied to this shorter derivation proves the claim.

The TAENCH and TAENCL cases are impossible, since the condition of the rule would then imply $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : \{T\}_{T'} \to c'$ for some T, T', c', which is not possible by point 2.

In the TENCH case, there exist T, T', c' such that $c = \{\mathtt{enc}(M_1, M_2) \sim N\} \cup c'$, $T' <: \mathrm{key}^{\mathtt{HH}}(T)$, and $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : (T)_{T'} \to c'$. By point 1, since $\mathtt{enc}(M_1, M_2)$ is not a variable, there exist N_1, N_2, c_1, c_2 such that $N = \mathtt{enc}(N_1, N_2)$, $c' = c_1 \cup c_2$, $\Gamma \vdash M_1 \sim N_1 : T \to c_1$, and $\Gamma \vdash M_2 \sim N_2 : T' \to c_2$. Hence by subtyping, $\Gamma \vdash M_2 \sim N_2 : \mathrm{key}^{\mathtt{HH}}(T) \to c_2$.

Finally in the TENCL case, there similarly exist T, T' such that $T <: \mathrm{key}^{\mathtt{LL}}(T')$ or $T = \mathtt{LL}$, and $\Gamma \vdash \mathtt{enc}(M_1, M_2) \sim N : (\mathtt{LL})_T \to c$. Hence by point 1, there exist N_1, N_2, c_1, c_2 such that $N = \mathtt{enc}(N_1, N_2)$, $c = c_1 \cup c_2$ $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$, and $\Gamma \vdash M_2 \sim N_2 : T \to c_2$. Hence in any case (potentially using subtyping), $\Gamma \vdash M_2 \sim N_2 : \mathtt{LL} \to c_2$, which proves the claim.

Point 6 has a similar proof to point 5.

The symmetric properties, as described in point 7, have analogous proofs. □

Lemma A.16 (Signature types)**.** *For all environment* Γ*, type* T*, messages* M_1*,* M_2*,* N*, and set of constraints* c*:*

1. *If* $T <: \mathtt{LL}$ *and* $\Gamma \vdash \mathtt{sign}(M_1, M_2) \sim N : T \to c$ *then* $T = \mathtt{LL}$*.*
2. *If* $\Gamma \vdash \mathtt{sign}(M_1, M_2) \sim N : \mathtt{LL} \to c$ *then there exist* N_1*,* N_2 *such that* $N = \mathtt{sign}(N_1, N_2)$*, and*
   - *either there exist* T*,* c' *and* c'' *such that* $\Gamma \vdash M_2 \sim N_2 : \mathrm{eqkey}^{\mathtt{HH}}(T) \to \emptyset$*,* $c = \{\mathtt{sign}(M_1, M_2) \sim N\} \cup c' \cup c''$*,* $\Gamma \vdash M_1 \sim N_1 : T \to c'$*, and* $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c''$*;*
   - *or there exists* c_1*,* c_2 *such that* $c = c_1 \cup c_2$*,* $\Gamma \vdash M_2 \sim N_2 : \mathtt{LL} \to c_2$ *and* $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$*.*
3. *The symmetric properties to the previous points,* i.e. *when the term on the right is a signature, also hold.*

Proof. We prove point 1 by induction on the proof of $\Gamma \vdash \mathtt{sign}(M, k) \sim N : T \to c$. Because of the form of the terms, the last rule applied can only be THIGH, TOR, TENCH, TENCL, TAENCH, TAENCL, TSIGNH, TSIGNL, TLR', TLRL' or TSUB.

The THIGH, TLR', TOR cases are actually impossible by Lemma A.1, since $T <: \mathtt{LL}$. In the TSUB case, we have $\Gamma \vdash \mathtt{sign}(M_1, M_2) \sim N : T' \to c$ for some T' such that $T' <: T$. By transitivity of $<:$, $T' <: \mathtt{LL}$, and the induction hypothesis proves the claim. In all other cases, $T = \mathtt{LL}$ and the claim holds.

We prove point 2 by induction on the proof of $\Gamma \vdash \mathtt{sign}(M_1, M_2) \sim N : \mathtt{LL} \to c$. Because of the form of the terms and of the type (*i.e.* $\mathtt{LL}$) the last rule applied can only be TENCH, TENCL, TAENCH, TAENCL, TSIGNH, TSIGNL, TLRL' or TSUB.

The TLRL' case is impossible, since by Lemma A.14 it would imply that $\mathtt{sign}(M_1, M_2)$ is a variable or a nonce.

In the TSUB case, we have $\Gamma \vdash \mathtt{sign}(M_1, M_2) \sim N : T \to c$ for some T such that $T <: \mathtt{LL}$. By point 1, $T = \mathtt{LL}$, and the premise of the rule thus gives a shorter derivation of $\Gamma \vdash \mathtt{sign}(M_1, M_2) \sim N : \mathtt{LL} \to c$. The induction hypothesis applied to this shorter derivation proves the claim.

The TENCH, TENCL, TAENCH and TAENCL cases are impossible, since the condition of the rule would then imply $\Gamma \vdash \mathtt{sign}(M_1, M_2) \sim N : (T)_{T'} \to c'$ (or $\{T\}_{T'}$) for some T, T', c', which is not possible by Lemma A.15.

Finally, in the TSIGNH and TSIGNL cases, the premises of the rule directly proves the claim.

The symmetric properties, as described in point 3, have analogous proofs. □

Lemma A.17 (Pair types)**.** *For all environment* Γ*, for all* M*,* N*,* T*,* c*:*

1. *For all* T_1*,* T_2*, if* $\Gamma \vdash M \sim N : T_1 * T_2 \to c$ *then*
   - *either there exist* M_1*,* M_2*,* N_1*,* N_2*,* c_1*,* c_2 *such that* $M = \langle M_1, M_2 \rangle$*,* $N = \langle N_1, N_2 \rangle$*,* $c = c_1 \cup c_2$*, and* $\Gamma \vdash M_1 \sim N_1 : T_1 \to c_1$ *and* $\Gamma \vdash M_2 \sim N_2 : T_2 \to c_2$*;*
   - *or* M *and* N *are variables.*
2. *For all* M_1*,* M_2*, if* $T <: \mathtt{LL}$ *and* $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : T \to c$ *then either* $T = \mathtt{LL}$ *or there exists* T_1*,* T_2 *such that* $T = T_1 * T_2$*.*
3. *For all* M_1*,* M_2*, if* $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : \mathtt{LL} \to c$ *then there exist* N_1*,* N_2*,* c_1*,* c_2*, such that* $c = c_1 \cup c_2$*,* $N = \langle N_1, N_2 \rangle$*,* $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$ *and* $\Gamma \vdash M_2 \sim N_2 : \mathtt{LL} \to c_2$*.*
4. *The symmetric properties to the previous two points (*i.e. *when the term on the right is a pair) also hold.*

Proof. Let us prove point 1 by induction on the typing derivation $\Gamma \vdash M \sim N : T_1 * T_2 \to c$. Because of the form of the type, and by well-formedness of Γ, the only possibilities for the last rule applied are TVAR, TPAIR, and TSUB.

The claim clearly holds in the TVAR and TPAIR cases.

In the TSUB case, $\Gamma \vdash M \sim N : T' \to c$ for some $T' <: T_1 * T_2$, and by Lemma A.1, $T' = T'_1 * T'_2$ for some T'_1, T'_2 such that $T'_1 <: T_1$ and $T'_2 <: T_2$. Therefore, by applying the induction hypothesis to $\Gamma \vdash M \sim N : T'_1 * T'_2 \to c$, M and N are either two variables, and the claim holds; or two pairs, *i.e.* there exist M_1, M_2, N_1, N_2, c_1, c_2 such that $M = \langle M_1, M_2 \rangle$, $N = \langle N_1, N_2 \rangle$, $c = c_1 \cup c_2$, and for $i \in \{1, 2\}$, $\Gamma \vdash M_i \sim N_i : T'_i \to c_i$. Hence, by subtyping, $\Gamma \vdash M_i \sim N_i : T_i \to c_i$, and the claim holds.

We now prove point 2 by induction on the proof of $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : T \to c$. Because of the form of the terms, the last rule applied can only be THIGH, TOR, TPAIR, TENCH, TENCL, TAENCH, TAENCL, TLR', TLRL' or TSUB.

The THIGH, TLR', and TOR cases are actually impossible by Lemma A.1, since $T <: \mathtt{LL}$.

The TLRL' and case is also impossible, since by Lemma A.14 it would imply that $\langle M_1, M_2 \rangle$ is either a variable or a nonce.

The TENCH, TENCL, TAENCH, TAENCL cases are impossible, since the condition of the rule would then imply $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : (T)_{T'} \to c'$ (or $\{T\}_{T'}$) for some T, T', c', which is not possible by Lemma A.15.

In the TPAIR case, the claim clearly holds.

Finally, in the TSUB case, we have $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : T' \to c$ for some T' such that $T' <: T$. By transitivity of $<:$, $T' <: \mathtt{LL}$, and we may apply the induction hypothesis to $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : T' \to c$. Hence either $T' = \mathtt{LL}$ or $T' = T'_1 * T'_2$ for some T'_1, T'_2. By Lemma A.1, this implies in the first case that $T = \mathtt{LL}$ and in the second case that $T = \mathtt{LL}$ or T is also a pair type ($T \neq \mathtt{HL}$ and $T \neq \mathtt{HH}$ in both cases, since we already know that $T <: \mathtt{LL}$).

We prove point 3 as a consequence of the first two points, by induction on the derivation of $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : \mathtt{LL} \to c$. The last rule in this derivation can only be TENCH, TENCL, TAENCH, TAENCL, TLR', TLRL' or TSUB by the form of the types and terms, but similarly to the previous point TENCH, TENCL, TAENCH, TAENCL, TLR' and TLRL' are actually not possible.

Hence the last rule of the derivation is TSUB. We have $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : T \to c$ for some T such that $T <: \mathtt{LL}$. By point 2, either $T = \mathtt{LL}$ or there exist T_1, T_2 such that $T = T_1 * T_2$. If $T = \mathtt{LL}$, we have a shorter proof of $\Gamma \vdash \langle M_1, M_2 \rangle \sim N : \mathtt{LL} \to c$ and we conclude by the induction hypothesis. Otherwise, since $T <: \mathtt{LL}$, by Lemma A.1, $T_1 <: \mathtt{LL}$ and $T_2 <: \mathtt{LL}$. Moreover by the first property, there exist N_1, N_2, c_1, c_2 such that $N = \langle N_1, N_2 \rangle$, $c = c_1 \cup c_2$, $\Gamma \vdash M_1 \sim N_1 : T_1 \to c_1$, and $\Gamma \vdash M_2 \sim N_2 : T_2 \to c_2$.

Thus by subtyping, $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$ and $\Gamma \vdash M_2 \sim N_2 : \mathtt{LL} \to c_2$, which proves the claim.

The symmetric properties, as described in point 4, have analogous proofs.

□

Lemma A.18 (Type for keys, nonces and constants)**.** *For all well-formed environment Γ, for all messages M, N, for all key $k \in \mathcal{K}$, for all nonce or constant $n \in \mathcal{N} \cup \mathcal{C}$, for all c, l, the following properties hold:*

1. *For all T, T', if $\Gamma \vdash M \sim N : T \to c$ and $T <: \mathrm{key}^l(T')$, then $c = \emptyset$; and either M, N are in $\mathcal{BK}$ and $\Gamma(M, N) <: T$; or M and N are variables.*
2. *If $\Gamma \vdash M \sim N : \mathrm{pkey}(T) \to c$, then $c = \emptyset$; and either $\exists M', N'.M = \mathtt{pk}(M') \wedge N = \mathtt{pk}(N') \wedge \Gamma \vdash M' \sim N' : T \to \emptyset$; or M, N are variables.*
3. *If $\Gamma \vdash M \sim N : \mathrm{vkey}(T) \to c$, then $c = \emptyset$; and either $\exists M', N'.M = \mathtt{vk}(M') \wedge N = \mathtt{vk}(N') \wedge \Gamma \vdash M' \sim N' : T \to \emptyset$; or M, N are variables.*
4. *If $l \in \{\mathtt{LL}, \mathtt{HH}\}$, and $\Gamma \vdash k \sim N : l \to c$, then $N \in \mathcal{K}$, $c = \emptyset$, and either $k, N \in \mathcal{BK}$ and there exists T such that $\Gamma(k, N) <: \mathrm{key}^l(T)$, or $l = \mathtt{LL}$, and $k = N \in \mathcal{FK}$.*
5. *If $\Gamma \vdash \mathtt{pk}(M') \sim N : \mathtt{LL} \to c$, then $c = \emptyset$, $\exists N'.N = \mathtt{pk}(N')$, and either $\exists l, T'.T <: \mathrm{eqkey}^l(T') \wedge \Gamma \vdash M' \sim N' : T \to \emptyset$, or $\exists k \in \mathrm{keys}(\Gamma) \cup \mathcal{FK}.M' = N' = k$.*

6. *If* $\Gamma \vdash \mathtt{vk}(M') \sim N : \mathtt{LL} \to c$, *then* $c = \emptyset$, $\exists N'.N = \mathtt{vk}(N')$, *and either* $\exists l, T'.T <: \mathrm{eqkey}^l(T') \wedge \Gamma \vdash M' \sim N' : T \to \emptyset$, *or* $\exists k \in \mathrm{keys}(\Gamma) \cup \mathcal{FK}.M' = N' = k$.

7. *If* $\Gamma \vdash n \sim N : \mathtt{HH} \to c$, *then* $n \in \mathcal{BN}$, $c = \emptyset$ *and either* $\Gamma(n) = \tau_n^{\mathtt{HH},1}$ *or* $\tau_n^{\mathtt{HH},\infty}$.

8. *If* $\Gamma \vdash n \sim N : \mathtt{LL} \to c$, *then* $N = n$, $c = \emptyset$, *and either there exists* $a \in \{1, \infty\}$ *such that* $\Gamma(n) = \tau_n^{\mathtt{LL},a}$, *or* $n \in \mathcal{FN} \cup \mathcal{C}$.

9. *The symmetric properties to points 4 to 8* (i.e. *with* k *(resp.* $\mathtt{pk}(M')$, $\mathtt{vk}(M')$, n*) on the right) also hold.*

Proof. Point 1 is easily proved by induction on the derivation of $\Gamma \vdash M \sim N : T \to c$. Indeed, by the form of the type, using Lemma A.1, the last rule can only be TKey, TVar, or TSub. In the TKey and TVar cases the claim clearly holds. In the TSub case, we have $T'' <: T$ such that $\Gamma \vdash M \sim N : T' \to c$ with a shorter derivation. By transitivity, $T'' <: \mathrm{key}^l(T')$. Thus by applying by the induction hypothesis, either M, N are variables, or they are keys and $\Gamma(M, N) <: T'' <: T$, and in both cases the claim holds.

Similarly, we prove point 2 by induction on the derivation of $\Gamma \vdash M \sim N : \mathrm{pkey}(T) \to c$. Indeed, by the form of the type, and since Γ is well-formed, the last rule can only be TPubKey, TVar, or TSub. In the TPubKey and TVar cases the claim clearly holds. In the TSub case, we have $T' <: \mathrm{pkey}(T)$ such that $\Gamma \vdash M \sim N : T' \to c$ with a shorter derivation. By Lemma A.1, $T' = \mathrm{pkey}(T)$. We conclude the proof by applying by the induction hypothesis to the shorter derivation of $\Gamma \vdash M \sim N : T' \to c$.

Point 3 has a similar proof to point 2.

We prove point 4 by induction on the derivation of $\Gamma \vdash k \sim N : l \to c$. Because of the form of the terms and type, and by well-formedness of Γ, the last rule applied can only be TCstFN, TEncH, TEncL, TAencH, TAencL, TLR', TLRL' or TSub.

In the TCstFN case, $k = N \in \mathcal{FK}$ and the claim holds.

The TEncH, TEncL, TAencH, TAencL cases are impossible since they would imply that $\Gamma \vdash k \sim N : (T')_{k',k''} \to c'$ (or $\{T'\}_{k',k''}$) for some T', k', k'', c', which is impossible by Lemma A.15.

The TLR' and TLRL' cases are impossible. Indeed in these cases, we have $\Gamma \vdash k \sim N : [\![\tau_m^{l,a} ; \tau_n^{l',a}]\!] \to \emptyset$ for some m, n. Lemma A.14 then implies that $m = k$ (and $n = N$), which is contradictory.

Finally, in the TSub case, we have $\Gamma \vdash k \sim N : T \to c$ for some T such that $T <: l$. By Lemma A.1, this implies that T is either a pair type, an encryption type, a public or verification key type, a key type, or l. The pair, encryption, public and verification key cases are impossible, respectively by Lemma A.17, Lemma A.15, and point 2, since $k \in \mathcal{K}$. The last case is trivial by the induction hypothesis. Only the case where $T <: \mathrm{key}^l(T')$ (for some T') remains. By point 1, in that case, since k is not a variable, we have $N \in \mathcal{K}$ and $\Gamma(k, N) <: \mathrm{key}^l(T')$, and therefore the claim holds.

Similarly, we prove point 5 by induction on the derivation of $\Gamma \vdash \mathtt{pk}(M') \sim N : \mathtt{LL} \to c$. Because of the form of the terms and type, and by well-formedness of Γ, the last rule applied can only be TEncH, TEncL, TAencH, TAencL, TLRL', TPubKeyL or TSub.

The TEncH, TEncL, TAencH, TAencL cases are impossible since they would imply that $\Gamma \vdash \mathtt{pk}(M') \sim N : (T')_k \to c'$ (or $\{T'\}_{k'}$) for some T', k, c', which is impossible by Lemma A.15.

The TLRL' case is also impossible. Indeed in this case, we have $\Gamma \vdash \mathtt{pk}(M') \sim N : [\![\tau_m^{l,a} ; \tau_n^{l',a}]\!] \to \emptyset$ for some m, n. Lemma A.14 then implies that $m = \mathtt{pk}(M')$ (and $n = N$), which is contradictory.

In the TSub case, we have $\Gamma \vdash \mathtt{pk}(M') \sim N : T \to c$ for some T such that $T <: \mathtt{LL}$. By Lemma A.1, this implies that T is either a pair type, an encryption type, a public or verification key type, a key type, or LL. Just as in the previous point, the pair, encryption, verification key, and key cases are impossible. If $T = \mathtt{LL}$, the claim trivially holds by the induction hypothesis. The case where $T = \mathrm{pkey}(T')$ (for some T') remains. Since $\Gamma \vdash \mathtt{pk}(M') \sim N : T \to c$, by point 2 we have $N = \mathtt{pk}(N')$ for some N', $c = \emptyset$ and $\Gamma \vdash M' \sim N' : T' \to \emptyset$. In addition, by Lemma A.1, since $T <: \mathtt{LL}$, there exist l, T'' such that $T' <: \mathrm{eqkey}^l(T'')$, and the claim holds.

Finally in the TPubKeyL case, the claim clearly holds.

Point 6 has a similar proof to point 5.

The remaining properties have similar proofs to point 4. For point 7, *i.e.* if $\Gamma \vdash n \sim t : \mathtt{HH} \to c$, only the TNonce, TSub, and TLR' cases are possible. The claim clearly holds in the TNonce case.

In the TLR' case, we have $\Gamma \vdash n \sim t : [\![\tau_m^{\mathtt{HH},a} ; \tau_p^{\mathtt{HH},a}]\!] \to \emptyset$ for some m, p. Lemma A.14 then implies that $m = n$, and $p = t$, and $\Gamma(n) = \tau_m^{\mathtt{HH},a}$, and $\Gamma(p) = \tau_p^{\mathtt{HH},a}$, which proves the claim.

In the TSub case, $\Gamma \vdash n \sim t : T \to c$ for some $T <: \mathtt{HH}$, thus by Lemma A.1, T is either a pair type (impossible by Lemma A.17), an encryption type (impossible by Lemma A.15), a public key, verification key, or key type (impossible by points 1 to 3), or HH (and we conclude by the induction hypothesis).

For point 8, similarly, only the TNonceL, TCstFN, TSub, TLRL' cases are possible. The TSub case is proved in the same way as for the third property. The TLRL' case is proved similarly to the previous point. Finally the claim clearly holds in the TNonceL and TCstFN cases.

The symmetric properties, as described in point 9, have analogous proofs. □

Lemma A.19 (Typable messages either reduce on both sides, or fail on both sides)**.** *For all (well-formed)* Γ*, for all messages* M*,* M'*, for all* T*,* c*, if*

$$\Gamma \vdash M \sim N : T \to c,$$

then

$$M{\downarrow} = \bot \iff N{\downarrow} = \bot.$$

Proof. Note that since messages do not contain destructors, it is clear that for any message M, either $M{\downarrow} = M$ or $M{\downarrow} = \bot$.

We prove the lemma by induction on the type derivation for $\Gamma \vdash M \sim N : T \to c$.

In the TNonce, TNonceL, TCstFN, TPubKeyL, TVKeyL, TKey, TVar, TLR^1, TLR$^\infty$, TLRVar, it is clear that $M{\downarrow} = M \neq \bot$ and $N{\downarrow} = N \neq \bot$.

In the THigh case, the premise of the rule implies that $M{\downarrow} \neq \bot$ and $N{\downarrow} \neq \bot$.

In the TLR' and TLRL' cases, there exist l, l', a, m, n, c such that $\Gamma \vdash M \sim N : [\![\tau_m^{l,a} ; \tau_n^{l',a}]\!] \to c$. Hence by Lemma A.14, either M, N are variables, or $M = m$ and $N = n$. Either way, $M{\downarrow} = M \neq \bot$ and $N{\downarrow} = N \neq \bot$.

The TEnc case is more involved. In that case, there exist M', M'', N', N'', T', T'' such that $M = \mathtt{enc}(M', M'')$, $N = \mathtt{enc}(N', N'')$, and $T = (T')_{T''}$. Let us show that $M{\downarrow}\neq \bot \Rightarrow N{\downarrow}\neq \bot$ (the other direction has a similar proof). Since $M{\downarrow}\neq \bot$, we have $M'{\downarrow}\neq \bot$. Hence, as $\Gamma \vdash M' \sim N' : T' \to c'$ for some c', by the induction hypothesis, $N'{\downarrow}\neq \bot$. Since $M{\downarrow}\neq \bot$, we also have $M''{\downarrow}\in \mathcal{K}$, *i.e.* by the previous remark, $M'' \in \mathcal{K}$. In addition, $\Gamma \vdash M'' \sim N'' : T'' \to c''$ for some c'', and either $T'' = \mathtt{LL}$ or $\exists l, T'''.T'' <: \mathrm{key}^{l}(T''')$. Hence, by Lemma A.18, either $M'', N'' \in \mathcal{K}$, or $M'', N'' \in \mathcal{X}$. The second case is impossible since $M'' \in \mathcal{K}$. We thus have $N'{\downarrow}\neq \bot$ and $N''{\downarrow}\in \mathcal{K}$. Therefore, $N{\downarrow}\neq \bot$ and the claim holds.

The TAenc, TPubKey, TVKey, TSignH, TSignL cases are similar to the TEnc case.

In the remaining cases, *i.e.* TPair, TEncH, TEncL, TAencH, TAencL, THash, THashL, TSub, and TOr, the claim directly follows from the induction hypothesis. We write the proof for the TPair case. In that case there exist M', M'', N', N'', T', T'' such that $M = \langle M', M''\rangle$, $N = \langle N', N''\rangle$, and $T = T' * T''$. Since $\Gamma \vdash M' \sim N' : T' \to c'$ for some c', by the induction hypothesis, $M'{\downarrow}= \bot \iff N'{\downarrow}= \bot$. Similarly, $M''{\downarrow}= \bot \iff N''{\downarrow}= \bot$. Therefore, $M{\downarrow}= \bot \iff N{\downarrow}= \bot$, and the claim holds. □

Lemma A.20 (Application of destructors)**.** *For all Γ, for all t, t', T, c, for all ground substitutions σ, σ' such that $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma') = \mathrm{vars}(t) \cup \mathrm{vars}(t')$, if $\Gamma \vdash t \sim t' : T$ and $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c$, where $\Gamma' = \Gamma_{\mathcal{N},\mathcal{K}} \cup \Gamma|_{\mathrm{dom}(\sigma)}$, then:*

1. *We have:*
$$(t\sigma){\downarrow}= \bot \iff (t'\sigma'){\downarrow}= \bot$$
2. *And if $(t\sigma){\downarrow}\neq \bot$ then there exists $c' \subseteq c$ such that*
$$\Gamma_{\mathcal{N},\mathcal{K}} \vdash (t\sigma){\downarrow} \sim (t'\sigma'){\downarrow} : T \to c'$$

Proof. Since $\Gamma \vdash t \sim t' : T$, by examining the typing rules for destructors, we can distinguish four cases for t, t'.

- $t = \mathtt{dec}(x, M)$ and $t' = \mathtt{dec}(x, M)$, for some variable $x \in \mathcal{X}$, and some $M \in \mathcal{X} \cup \mathcal{K}$. We know that $\Gamma \vdash t \sim t' : T$, which can be proved using the rule DDecL, DDecT, DDecT', DDecH', or DDecL'. In the DDecL, DDecH', DDecL' cases, $\Gamma(x) = \mathtt{LL}$, and in the DDecT and DDecT' cases $\Gamma(x) = (T)_{T'}$ for some T'. By assumption we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$ for some $c_x \subseteq c$.
  - Let us prove 1) by contraposition. Assume $t\sigma{\downarrow}\neq \bot$. Hence, $\sigma(x) = \mathtt{enc}(M', M\sigma)$ for some M', and $M\sigma$ is a key $k \in \mathcal{K}$. We will now show that $M\sigma' = M\sigma = k$. It is clear if $M \in \mathcal{K}$. Otherwise, $M \in \mathcal{X}$, and:
    - In the DDecL and DDecL' cases, $\Gamma \vdash M \sim M : \mathtt{LL} \to \emptyset$. Since σ, σ' are well-typed, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma \sim M\sigma' : \mathtt{LL} \to c''$ for some c'', *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash k \sim M\sigma' : \mathtt{LL} \to c''$. Thus by Lemma A.18, either $M\sigma' = k \in \mathcal{FK}$; or $M\sigma' \in \mathcal{BK}$ and $\Gamma(k, M\sigma') <: \mathrm{key}^{\mathtt{LL}}(T''')$ for some T'''. Hence, in either case (using the well-formedness of Γ in the second case), $M\sigma' = k$.
    - In the DDecH', DDecT, DDecT' cases, $\Gamma(M) = \mathrm{seskey}^{l,a}(T''')$ for some l, T'''. Hence, $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma \sim M\sigma' : \mathrm{seskey}^{l,a}(T''') \to c''$ for some c''. Therefore, by Lemma A.18 (and since $M\sigma$, $M\sigma'$ are ground), $M\sigma' \in \mathcal{K}$, and $\Gamma(k, M\sigma') <: \mathrm{seskey}^{l,a}(T''')$. Thus, by Lemma A.1, $\Gamma(k, M\sigma') = \mathrm{seskey}^{l,a}(T''')$, and since Γ is well-formed, $M\sigma' = k$.

Let us now show that there exists a ground message N' such that $\sigma'(x) = \mathtt{enc}(N', k)$.

* In the DDecL, DDecH' and DDecL' cases, $\Gamma(x) = \mathtt{LL}$. Since σ, σ' are well-typed, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c''$ for some c'', *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{enc}(M', k) \sim \sigma'(x) : \mathtt{LL} \to c''$. Thus by Lemma A.15, there exist N, N' such that $\sigma'(x) = \mathtt{enc}(N', N)$. In addition:
  - either $\Gamma_{\mathcal{N},\mathcal{K}} \vdash k \sim N : \mathrm{key}^{\mathtt{HH}}(T''') \to c'''$ for some T''', c''': in that case, by Lemma A.18, $N \in \mathcal{K}$ and $\Gamma(k, N) <: \mathrm{key}^{\mathtt{HH}}(T''')$. We have already shown that $\Gamma(k, k)$ is either a subtype of $\mathtt{LL}$, or $\mathrm{seskey}^{l,a}(T'''')$ for some l, T''''. By well-formedness of Γ, only the second case is possible, and it implies that $N = k$.
  - or $\Gamma_{\mathcal{N},\mathcal{K}} \vdash k \sim N : \mathtt{LL} \to c'''$ for some c''': in that case, by Lemma A.18, $N = k$.

  In any case we have $\sigma'(x) = \mathtt{enc}(N', k)$.
* In the DDecT and DDecT' cases, $\Gamma(x) = (T)_{\mathrm{seskey}^{l,a}(T'')}$ and $\Gamma \vdash M \sim M : \mathrm{seskey}^{l,a}(T'') \to \emptyset$ for some l, T''. Hence we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : (T)_{\mathrm{seskey}^{l,a}(T'')} \to c''$ for some c'', *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{enc}(M', k) \sim \sigma'(x) : (T)_{\mathrm{seskey}^{l,a}(T'')} \to c''$. Therefore, by Lemma A.15, there exist N, N' such that $\sigma'(x) = \mathtt{enc}(N', N)$, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash k \sim N : \mathrm{seskey}^{l,a}(T'') \to \emptyset$. Thus, by Lemma A.18, $N \in \mathcal{K}$ and $\Gamma(k, N) = \mathrm{seskey}^{l,a}(T'')$. By well-formedness of Γ, this implies that $N = k$. Therefore $\sigma'(x) = \mathtt{enc}(N', k)$.

Hence, in any case, $t'\sigma' = \mathtt{dec}(\mathtt{enc}(N', k), k)$. By assumption, σ, σ' are well-typed, and $\sigma(x)\!\downarrow \neq \bot$. Thus by Lemma A.19 $\sigma'(x)\!\downarrow \neq \bot$. Then $N'\!\downarrow \neq \bot$. Therefore we have $t'\sigma'\!\downarrow = N'\!\downarrow \neq \bot$, which proves the first direction of 1). The other direction is analogous.

– Moreover, still assuming $t\sigma\!\downarrow \neq \bot$, and keeping the notations from the previous point, we have $t\sigma\!\downarrow = M'$ and $t'\sigma'\!\downarrow = N'$. The destructor typing rule applied to prove $\Gamma \vdash t \sim t' : T$ can be DDecT, DDecT', DDecL, DDecH', or DDecL'.

  * In the DDecT and DDecT' cases, $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : (T)_{\mathrm{seskey}^{l,a}(T'')} \to c''$ for some $c'' \subseteq c$, *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{enc}(M', k) \sim \mathtt{enc}(N', k) : (T)_{\mathrm{seskey}^{l,a}(T'')} \to c''$. Thus, by Lemma A.15, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M' \sim N' : T \to c'''$ for some $c''' \subseteq c''$, and the claim holds.
  * In the DDecL and DDecL' cases, we have $T = \mathtt{LL}$, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c''$ for some $c'' \subseteq c$, *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{enc}(M', k) \sim \mathtt{enc}(N', k) : \mathtt{LL} \to c''$. In addition we have $\Gamma \vdash M \sim M : \mathtt{LL} \to \emptyset$, and thus $\Gamma \vdash k \sim k : \mathtt{LL} \to c'''$ for some c'''. Therefore, by Lemma A.15, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M' \sim N' : \mathtt{LL} \to c'''$ for some $c''' \subseteq c''$ (the case where $\Gamma \vdash k \sim k : \mathrm{key}^{\mathtt{HH}}(T'') \to c'''$ is impossible by Lemma A.18, since $\Gamma \vdash k \sim k : \mathtt{LL} \to c'''$), and the claim holds.
  * In the DDecH' case, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c''$ for some $c'' \subseteq c$, *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{enc}(M', k) \sim \mathtt{enc}(N', k) : \mathtt{LL} \to c''$. In addition we have $\Gamma \vdash M \sim M : \mathrm{seskey}^{\mathtt{HH},a}(T) \to \emptyset$, and thus $\Gamma \vdash k \sim k : \mathrm{seskey}^{\mathtt{HH},a}(T) \to c'''$ for some c'''. Therefore, by Lemma A.15, there exists $c'''' \subseteq c''$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M' \sim N' : T \to c''''$ (the case where $\Gamma \vdash k \sim k : \mathtt{LL} \to c''''$ is impossible by Lemma A.18, since $\Gamma \vdash k \sim k : \mathrm{seskey}^{\mathtt{HH},a}(T) \to c'''$), and the claim holds.

  In all cases, point 2) holds, which concludes this case.

- $t = \mathtt{adec}(x, M)$ and $t' = \mathtt{adec}(x, M)$, for some variable $x \in \mathcal{X}$, and some $M \in \mathcal{X} \cup \mathcal{K}$. We know that $\Gamma \vdash t \sim t' : T$, which can be proved using the rule DAdecL, DAdecT, DAdecT', DAdecH', or DAdecL'. In the DAdecL, DAdecH', DAdecL' cases, $\Gamma(x) = \mathtt{LL}$, and in the DAdecT and DAdecT' cases $\Gamma(x) = \{T\}_{T'}$ for some T'. By assumption we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$ for some $c_x \subseteq c$.

  - Let us prove 1) by contraposition. Assume $t\sigma{\downarrow} \neq \bot$. Hence, $\sigma(x) = \mathtt{aenc}(M', \mathtt{pk}(M\sigma))$ for some M', and $M\sigma$ is a key $k \in \mathcal{K}$. We will now show that $M\sigma' = M\sigma = k$. It is clear if $M \in \mathcal{K}$. Otherwise, $M \in \mathcal{X}$, and:

    * In the DAdecL and DAdecL' cases, $\Gamma \vdash M \sim M : \mathtt{LL} \to \emptyset$. Since σ, σ' are well-typed, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma \sim M\sigma' : \mathtt{LL} \to c''$ for some c'', *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash k \sim M\sigma' : \mathtt{LL} \to c''$. Thus by Lemma A.18, either $M\sigma' = k \in \mathcal{FK}$; or $M\sigma' \in \mathcal{BK}$, and $\Gamma(k, M\sigma') <: \mathrm{key}^{\mathtt{LL}}(T''')$ for some T'''. Hence, in any case (by well-formedness of Γ in the case where $M\sigma' \in \mathcal{BK}$), $M\sigma' = k$.
    * In the DAdecH', DAdecT, DAdecT' cases, $\Gamma(M) = \mathrm{seskey}^{l,a}(T''')$ for some l, T'''. Hence, $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma \sim M\sigma' : \mathrm{seskey}^{l,a}(T''') \to c''$ for some c''. Therefore, by Lemma A.18 (and since $M\sigma$, $M\sigma'$ are ground), $M\sigma' \in \mathcal{K}$, and $\Gamma(k, M\sigma') <: \mathrm{seskey}^{l,a}(T''')$. Thus, by Lemma A.1, $\Gamma(k, M\sigma') = \mathrm{seskey}^{l,a}(T''')$, and since Γ is well-formed, $M\sigma' = k$.

    Let us now show that $\sigma'(x) = \mathtt{aenc}(N', \mathtt{pk}(k))$ for some ground N'.

    * In the DAdecL, DAdecH' and DAdecL' cases, $\Gamma(x) = \mathtt{LL}$. Since σ, σ' are well-typed, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c''$ for some c'', *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{aenc}(M', \mathtt{pk}(k)) \sim \sigma'(x) : \mathtt{LL} \to c''$. Thus by Lemma A.15, there exist N, N' such that $\sigma'(x) = \mathtt{aenc}(N', N)$. In addition:
      · either $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{pk}(k) \sim N : \mathrm{pkey}(T''') \to c'''$ and $T''' <: \mathrm{key}^{\mathtt{HH}}(T'''')$ for some T''', T'''', c''': in that case, by Lemma A.18, $N = \mathtt{pk}(k')$ for some key $k' \in \mathcal{K}$ such that $\Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T''')$. We have already shown that $\Gamma(k, k)$ is either a subtype of $\mathtt{LL}$, or $\mathrm{seskey}^{l,a}(T'''')$ for some l, T''''. By well-formedness of Γ, only the second case is possible, and it implies that $k' = k$.
      · or $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{pk}(k) \sim N : \mathtt{LL} \to c'''$ for some c''': in that case, by Lemma A.18, $N = \mathtt{pk}(k)$.

      In any case we have $\sigma'(x) = \mathtt{aenc}(N', \mathtt{pk}(k))$.
    * In the DAdecT and DAdecT' cases, we have $\Gamma(x) = \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T''))}$ as well as $\Gamma \vdash M \sim M : \mathrm{seskey}^{l,a}(T'') \to \emptyset$ for some l, T''. Hence we have

    $$\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T''))} \to c''$$

    for some c'', *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{aenc}(M', \mathtt{pk}(k)) \sim \sigma'(x) : \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T''))} \to c''$. Therefore, by Lemma A.15, there exist N, N', c''' such that $\sigma'(x) = \mathtt{aenc}(N', N)$, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{pk}(k) \sim N : \mathrm{pkey}(\mathrm{seskey}^{l,a}(T'')) \to c'''$. Thus, by Lemma A.18, $N = \mathtt{pk}(k')$ for some key $k' \in \mathcal{K}$ and $\Gamma(k, k') = \mathrm{seskey}^{l,a}(T'')$. By well-formedness of Γ, this implies that $k' = k$. Therefore $\sigma'(x) = \mathtt{aenc}(N', \mathtt{pk}(k))$.

    Hence, in any case, $t'\sigma' = \mathtt{adec}(\mathtt{aenc}(N', \mathtt{pk}(k)), k)$, By assumption, σ, σ' are well-typed, and $\sigma(x){\downarrow} \neq \bot$. Thus by Lemma A.19 $\sigma'(x){\downarrow} \neq \bot$. Then $N'{\downarrow} \neq \bot$. Therefore we have $t'\sigma'{\downarrow} = N'{\downarrow} \neq \bot$, which proves the first direction of 1). The other direction is analogous.

– Moreover, still assuming $t\sigma\downarrow \neq \bot$, and keeping the notations from the previous point, we have $t\sigma\downarrow = M'$ and $t'\sigma'\downarrow = N'$. The destructor typing rule applied to prove $\Gamma \vdash t \sim t' : T$ can be DADECT, DADECT', DADECL, DADECH', or DADECL'.

∗ In the DADECT and DADECT' cases, we have

$$\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T''))} \to c''$$

for some $c'' \subseteq c$, *i.e.*

$$\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{aenc}(M', \mathtt{pk}(k)) \sim \mathtt{aenc}(N', \mathtt{pk}(k)) : \{T\}_{\mathrm{pkey}(\mathrm{seskey}^{l,a}(T''))} \to c''.$$

Thus, by Lemma A.15, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M' \sim N' : T \to c'''$ for some $c''' \subseteq c''$, and the claim holds.

∗ In the DADECL and DADECL' cases, we have $T = \mathtt{LL}$, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c''$ for some $c'' \subseteq c$, *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{aenc}(M', \mathtt{pk}(k)) \sim \mathtt{aenc}(N', \mathtt{pk}(k)) : \mathtt{LL} \to c''$. In addition we have $\Gamma \vdash M \sim M : \mathtt{LL} \to \emptyset$, and thus $\Gamma \vdash k \sim k : \mathtt{LL} \to c'''$ for some c'''. Therefore, by Lemma A.15, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M' \sim N' : \mathtt{LL} \to c''''$ for some $c'''' \subseteq c''$ (the case where $\Gamma \vdash k \sim k : \mathrm{key}^{\mathtt{HH}}(T'') \to c''''$ is impossible by Lemma A.18, since we already know that $\Gamma \vdash k \sim k : \mathtt{LL} \to c'''$), and the claim holds.

∗ In the DADECH' case, we have $T = T' \vee \mathtt{LL}$ for some type T'. In addition we know that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c''$ for some $c'' \subseteq c$, *i.e.*

$$\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{aenc}(M', \mathtt{pk}(k)) \sim \mathtt{aenc}(N', \mathtt{pk}(k)) : \mathtt{LL} \to c''.$$

In addition, $\Gamma \vdash M \sim M : \mathrm{seskey}^{\mathtt{HH},a}(T') \to \emptyset$, and thus there exists c''' such that $\Gamma \vdash k \sim k : \mathrm{seskey}^{\mathtt{HH},a}(T') \to c'''$. Therefore, by Lemma A.15, we know that

· either there exist types T'', T''', and constraints $c'''', c''''' \subseteq c''$ such that T'' is a subtype of $\mathrm{key}^{\mathtt{HH}}(T''')$, $\Gamma \vdash \mathtt{pk}(k) \sim \mathtt{pk}(k) : \mathrm{pkey}(T'') \to c''''$, and $\Gamma \vdash M' \sim N' : T''' \to c''''$. Since $\Gamma \vdash \mathtt{pk}(k) \sim \mathtt{pk}(k) : \mathrm{pkey}(T'') \to \emptyset$, by Lemma A.18, we have $\Gamma(k,k) <: \mathrm{key}^{\mathtt{HH}}(T''')$. As we already know that $\Gamma \vdash k \sim k : \mathrm{seskey}^{\mathtt{HH},a}(T') \to c'''$, by the same lemma and Lemma A.1, we have $T''' = T'$. Thus $\Gamma \vdash M' \sim N' : T' \to c''''$, and by rule TOR, we have $\Gamma \vdash M' \sim N' : T' \vee \mathtt{LL} \to c''''$.

· or $\Gamma \vdash M' \sim N' : \mathtt{LL} \to c''$, and by rule TOR we have $\Gamma \vdash M' \sim N' : T' \vee \mathtt{LL} \to c''''$.

In all cases, $\Gamma \vdash M' \sim N' : T \to c''''$ for some $c'''' \subset c''$, and point 2) holds, which concludes this case.

- $t = t' = \mathtt{checksign}(x, M)$. We know that $\Gamma \vdash t \sim t' : T$, which can be proved using either DCHECKH, DCHECKH', DCHECKL, or DCHECKL'. In both cases, $\Gamma(x) = \mathtt{LL}$. M is either a verification key or a variable. By assumption we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c_x$ for some $c_x \subseteq c$.

– Let us prove 1) by contraposition. Assume $t\sigma\downarrow \neq \bot$. Hence, $\sigma(x) = \mathtt{sign}(M', M''\sigma)$ for some M', M'', and $M''\sigma$ is a key $k \in \mathcal{K}$ such that $M\sigma = \mathtt{vk}(k)$.

We will now show that $M\sigma' = M\sigma = \mathtt{vk}(k)$. It is clear if M is a verification key. Otherwise, $M \in \mathcal{X}$, which means the rule applied to prove $\Gamma \vdash t \sim t' : T$ is DCHECKL'

or DCheckH'. In either case, from the form of the rule we have $\Gamma \vdash M \sim M : \mathtt{LL} \to \emptyset$, Since σ, σ' are well-typed, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma \sim M\sigma' : \mathtt{LL} \to c''$ for some c'', *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{vk}(k) \sim M\sigma' : \mathtt{LL} \to c''$. Thus by Lemma A.18, and since Γ is well-formed, $M\sigma' = \mathtt{vk}(k)$.

In addition, we know that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c_x$, *i.e.*

$$\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{sign}(M', k) \sim \sigma'(x) : \mathtt{LL} \to c_x.$$

Hence Lemma A.16 guarantees that there exist N', N'' such that $\sigma'(x) = \mathtt{sign}(N', N'')$, and either $\Gamma \vdash k \sim N'' : \mathrm{eqkey}^{\mathtt{HH}}(T') \to \emptyset$ for some T' or $\Gamma \vdash k \sim N'' : \mathtt{LL} \to c''$ for some c''. In both cases, Lemma A.18 implies that $N'' = k$. Thus we have $\sigma'(x) = \mathtt{sign}(N', k)$.

Hence, $t'\sigma' = \mathtt{checksign}(\mathtt{sign}(N', k), \mathtt{vk}(k))$, By assumption, σ, σ' are well-typed, and $\sigma(x)\!\downarrow \neq \bot$. Thus by Lemma A.19 $\sigma'(x)\!\downarrow \neq \bot$. Then $N'\!\downarrow \neq \bot$. Therefore we have $t'\sigma'\!\downarrow = N'\!\downarrow \neq \bot$, which proves the first direction of 1). The other direction is analogous.

– Moreover, still assuming $t\sigma\!\downarrow \neq \bot$, and keeping the notations from the previous point, we have $t\sigma\!\downarrow = M'$ and $t'\sigma'\!\downarrow = N'$. The destructor typing rule applied to prove $\Gamma \vdash t \sim t' : T$ can be DCheckH, DCheckH', DCheckL, or DCheckL'.

  * In the DCheckH and DCheckH' cases we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c_x$ for $c_x \subseteq c$, *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{sign}(M', k) \sim \mathtt{sign}(N', k) : \mathtt{LL} \to c_x$. In both cases we have $\Gamma \vdash \mathtt{vk}(k) \sim \mathtt{vk}(k) : \mathrm{vkey}(T') \to c''$ for some c'' and some $T' <: \mathrm{key}^{\mathtt{HH}}(T)$. Hence by Lemma A.18, $\Gamma(k, k) <: \mathrm{key}^{\mathtt{HH}}(T)$. By Lemma A.16, we know that there exist $c' \subseteq c_x$ such that $\Gamma \vdash M' \sim N' : T \to c'$ (the case where $\Gamma \vdash k \sim k : \mathtt{LL} \to c''$ is impossible by Lemma A.18 since $\Gamma(k, k) <: \mathrm{key}^{\mathtt{HH}}(T)$ and Γ is well-formed). This proves point 2).
  * In the DCheckL and DCheckL' cases we have $T = \mathtt{LL}$, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \mathtt{LL} \to c_x$ for $c_x \subseteq c$, *i.e.* $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \mathtt{sign}(M', k) \sim \mathtt{sign}(N', k) : \mathtt{LL} \to c_x$. By Lemma A.16, we know that there exist $c' \subseteq c_x$ such that $\Gamma \vdash M' \sim N' : \mathtt{LL} \to c'$. This proves point 2).

  In all cases, point 2) holds, which concludes this case.

• $t = t' = \pi_1(x)$. We know that $\Gamma \vdash t : t'T$, which can be proved using either rule DFst or DFstL. In the first case, $\Gamma(x) = T_1 * T_2$ is a pair type, and in the second case $\Gamma(x) = \mathtt{LL}$.

– We prove 1) by contraposition. Assume $t\sigma\!\downarrow \neq \bot$. Hence, $\sigma(x) = \langle M_1, M_2 \rangle$ for some M_1, M_2. By assumption σ, σ' are well-typed. Thus $\Gamma \vdash \sigma(x) \sim \sigma'(x) : \Gamma(x) \to c_x$, for some $c_x \subseteq c$. Thus, by applying Lemma A.17, in any case we know that there exist N_1, N_2 such that $N = \langle N_1, N_2 \rangle$. Since, σ, σ' are well-typed, and $\sigma(x)\!\downarrow \neq \bot$, by Lemma A.19, $\sigma'(x)\!\downarrow \neq \bot$. Then $N_1\!\downarrow \neq \bot$. Therefore we have $t'\sigma'\!\downarrow = N_1\!\downarrow \neq \bot$, which proves the first direction of 1). The other direction is analogous.

– Moreover, still assuming $t\sigma\!\downarrow \neq \bot$, and keeping the notations from the previous point, we have $t\sigma\!\downarrow = M_1$ and $t'\sigma'\!\downarrow = N_1$. In addition, we know that $\Gamma \vdash t : t'T$, which can be proved using either rule DFst or DFstL. Lemma A.17, which we applied in the previous point, also implies that there exist c_1, c_2, such that $c = c_1 \cup c_2$ and for $i \in \{1, 2\}$, $\Gamma \vdash M_i \sim N_i : T_i \to c_i$ (in the DFst case) or $\Gamma \vdash M_i \sim N_i : \mathtt{LL} \to c_i$ (in the DFstL case).

  We distinguish two cases for the rule applied to prove $\Gamma \vdash t : t'T$.

- DFST: Then $\Gamma(x) = T * T_2$ for some T_2, and $\Gamma \vdash M_1 \sim N_1 : T \to c_1(\subseteq c)$ proves 2).
- DFSTL: Then $T = \Gamma(x) = \mathtt{LL}$, and $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1(\subseteq c)$ proves 2).

In both cases, point 2) holds, which concludes this case.

- $t = t' = \pi_2(x)$. This case is similar to the previous one.

□

Lemma A.21 (LL type is preserved by attacker terms)**.** *For all (well-formed)* Γ*, for all frames* ϕ *and* ϕ' *with* $\Gamma \vdash \phi \sim \phi' : \mathtt{LL} \to c$*, for all attacker term* R *such that* $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi)$*, either there exists* $c' \subseteq c$ *such that*

$$\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c'$$

or

$$R\phi\downarrow = R\phi'\downarrow = \bot.$$

Proof. We show this property by induction over the attacker term R.

Induction Hypothesis: the statement holds for all subterms of R.

There are several cases for R. The base cases are the cases where R is a variable, a name in $\mathcal{FN}$ or a constant in $\mathcal{C}$.

1. $R = x$ Since $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi)$, we have $x \in \mathrm{dom}(\phi) = \mathrm{dom}(\phi')$, hence $R\phi = \phi(x)$. By assumption, there exists $c' \subseteq c$ such that $\Gamma \vdash \phi(x) \sim \phi'(x) : \mathtt{LL} \to c'$. Thus, by Lemma A.19, either $\phi(x) = \phi'(x) = \bot$, or $\phi(x)\downarrow = \phi(x)$ and $\phi'(x)\downarrow = \phi'(x)$. In the first case the claim clearly holds. In the second case, $R\phi\downarrow = \phi(x)$ and $R\phi'\downarrow = \phi'(x)$. Since $\Gamma \vdash \phi(x) \sim \phi'(x) : \mathtt{LL} \to c' \subseteq c$, the claim also holds.

2. $R = a$ with $a \in \mathcal{C} \cup \mathcal{FN} \cup \mathcal{FK}$. Then $R\phi\downarrow = R\phi'\downarrow = a$ and by rule TCSTFN, we have $\Gamma \vdash a \sim a : \mathtt{LL} \to \emptyset$. Hence the claim holds.

3. $R = \mathtt{pk}(K)$ We apply the induction hypothesis to K and distinguish three cases.
   (a) If $K\phi\downarrow = \bot$ then $K\phi'\downarrow = \bot$, hence $R\phi\downarrow = R\phi'\downarrow = \bot$.
   (b) If $K\phi\downarrow \neq \bot$ and is not a key then $K\phi'\downarrow \neq \bot$ (by IH), and by IH we have $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to c'$ for some $c' \subseteq c$. Then by Lemma A.18, $K\phi'\downarrow$ is not a key either. Hence $R\phi\downarrow = R\phi'\downarrow = \bot$.
   (c) If $K\phi\downarrow$ is a key, then by IH there exists $c' \subseteq c$ such that $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to c'$. Hence by Lemma A.18 (and since Γ is well-formed) $K\phi'\downarrow = K\phi'\downarrow \in \mathcal{K}$, and either $\Gamma(K\phi\downarrow, K\phi'\downarrow) <: \mathrm{key}^{\mathrm{LL}}(T)$ for some T, or $K\phi'\downarrow \in \mathcal{FK}$. Therefore by rule TPUBKEYL, $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to \emptyset$, and the claim holds.

4. $R = \mathtt{vk}(K)$ This case is analogous to the pk case.

5. $R = \langle R_1, R_2 \rangle$ where R_1 and R_2 are also attacker terms. We then apply the induction hypothesis to the same frames and R_1, R_2. We distinguish two cases:
   (a) $R_1\phi\downarrow = \bot \vee R_2\phi\downarrow = \bot$ In this case we also have $R_1\phi'\downarrow = \bot \vee R_2\phi'\downarrow = \bot$ and therefore $R\phi\downarrow = R\phi'\downarrow = \bot$.

(b) $R_1\phi\downarrow \neq \bot \wedge R_2\phi\downarrow \neq \bot$ In this case, by the induction hypothesis, we also have $R_1\phi'\downarrow \neq \bot \wedge R_2\phi'\downarrow \neq \bot$, and we also know that there exist $c_1 \subseteq c$ and $c_2 \subseteq c$ such that $\Gamma \vdash R_1\phi\downarrow \sim R_1\phi'\downarrow : \mathtt{LL} \to c_1$ and $\Gamma \vdash R_2\phi\downarrow \sim R_2\phi'\downarrow : \mathtt{LL} \to c_2$.
Thus, by the rule TPair followed by TSub, $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c_1 \cup c_2$. Since $c_1 \cup c_2 \subseteq c$, this proves the case.

6. $R = \mathtt{enc}(S, K)$ We apply the induction hypothesis to K and distinguish three cases.

   (a) If $K\phi\downarrow = \bot$ then $K\phi'\downarrow = \bot$, hence $R\phi\downarrow = R\phi'\downarrow = \bot$.

   (b) If $K\phi\downarrow \neq \bot$ and is not a key then $K\phi'\downarrow \neq \bot$ (by IH), and by IH we have $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to c'$ for some $c' \subseteq c$. Then, by Lemma A.18, $K\phi'\downarrow$ is not a key either. Hence $R\phi\downarrow = R\phi'\downarrow = \bot$.

   (c) If $K\phi\downarrow$ is a key, then by IH there exists $c' \subseteq c$ such that $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to c'$. Hence, by Lemma A.18 (and since Γ is well-formed), $K\phi'\downarrow = K\phi\downarrow \in \mathcal{K}$, and either $\Gamma(K\phi\downarrow, K\phi'\downarrow) <: \mathrm{key}^{\mathtt{LL}}(T)$ for some T, or $K\phi\downarrow \in \mathcal{FK}$. Thus, in each case, by rules TKey and TSub or TCstFN, $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to \emptyset$. We then apply the IH to S, and either $S\phi\downarrow = S\phi'\downarrow = \bot$, in which case $R\phi\downarrow = R\phi'\downarrow = \bot$; or there exists $c'' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c''$. Since $R\phi\downarrow = \mathtt{enc}(S\phi\downarrow, K\phi\downarrow)$, and similarly for ϕ', by rule TEnc, we have $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : (\mathtt{LL})_{\mathtt{LL}} \to c''$, and then by rule TEncL, $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c''$.

7. $R = \mathtt{aenc}(S, K)$ We apply the induction hypothesis to K and distinguish three cases.

   (a) If $K\phi\downarrow = \bot$ then $K\phi'\downarrow = \bot$, hence $R\phi\downarrow = R\phi'\downarrow = \bot$.

   (b) If $K\phi\downarrow \neq \bot$ and is not $\mathtt{pk}(k)$ for some $k \in \mathcal{K}$ then $K\phi'\downarrow \neq \bot$ (by IH), and by IH there exists $c' \subseteq c$ such that $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to c'$. Then, by Lemma A.18, $K\phi'\downarrow$ is not a public key either. Hence $R\phi\downarrow = R\phi'\downarrow = \bot$.

   (c) If $K\phi\downarrow = \mathtt{pk}(k)$ for some $k \in \mathcal{K}$, then by IH there exists $c' \subseteq c$ such that $\Gamma \vdash \mathtt{pk}(k) \sim K\phi'\downarrow : \mathtt{LL} \to c'$. Thus, by Lemma A.18, $K\phi'\downarrow = \mathtt{pk}(N)$ for some N such that either $N = k \in \mathrm{keys}(\Gamma) \cup \mathcal{FK}$; or $\Gamma \vdash k \sim N : \mathrm{eqkey}^{l}(T) \to c''$ for some l, T, c''. In the second case, the same lemma and the well-formedness of Γ also imply that $N = k$ and $k \in \mathrm{keys}(\Gamma)$. Thus, by rule TPubKeyL, $\Gamma \vdash \mathtt{pk}(k) \sim \mathtt{pk}(k) : \mathtt{LL} \to \emptyset$. We then apply the IH to S, and either $S\phi\downarrow = S\phi'\downarrow = \bot$, in which case $R\phi\downarrow = R\phi'\downarrow = \bot$; or there exists $c''' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c'''$. Therefore, by rule TAenc, $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \{\mathtt{LL}\}_{\mathtt{LL}} \to c''$, and by rule TAencL we have $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c''$.

8. $R = \mathtt{sign}(S, K)$ We apply the induction hypothesis to K and distinguish three cases.

   (a) If $K\phi\downarrow = \bot$ then $K\phi'\downarrow = \bot$, hence $R\phi\downarrow = R\phi'\downarrow = \bot$.

   (b) If $K\phi\downarrow \neq \bot$ and is not a key $k \in \mathcal{K}$ then $K\phi'\downarrow \neq \bot$ (by IH), and by IH we have $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to c'$ for some $c' \subseteq c$. Then, by Lemma A.18, $K\phi'\downarrow$ is not a key either. Hence $R\phi\downarrow = R\phi'\downarrow = \bot$.

   (c) If $K\phi\downarrow = k$ for some $k \in \mathcal{K}$, then by IH there exists $c' \subseteq c$ such that $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to c'$. Hence by Lemma A.18, (and since Γ is well-formed) $K\phi'\downarrow = k \in \mathcal{K}$ and either $\Gamma(k, k) <: \mathrm{key}^{\mathtt{LL}}(T)$ for some T, or $k \in \mathcal{FK}$. Thus, in either case, by rules TKey, TSub, and TCstFN, $\Gamma \vdash K\phi\downarrow \sim K\phi'\downarrow : \mathtt{LL} \to \emptyset$. We then apply the IH to S, and either $S\phi\downarrow = S\phi'\downarrow = \bot$, in which case $R\phi\downarrow = R\phi'\downarrow = \bot$; or there exists $c'' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c''$. Therefore by rule TSignL, $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c''$.

9. $R = \mathtt{h}(S)$ We apply the induction hypothesis to S. We distinguish two cases:
   (a) $S\phi\downarrow = \bot$ In this case we also have $S\phi'\downarrow = \bot$ and therefore $R\phi\downarrow = R\phi'\downarrow = \bot$.
   (b) $S\phi\downarrow \neq \bot$ In this case, by the induction hypothesis, we also have $S\phi'\downarrow \neq \bot$, and we also know that there exists $c' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c'$ Thus, by rule THASHL, $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c'$, which proves this case.
10. $R = \pi_1(S)$ We apply the induction hypothesis to S and distinguish three cases.
   (a) $S\phi\downarrow = \bot$ Then $S\phi'\downarrow = \bot$ (by IH), hence $R\phi\downarrow = R\phi'\downarrow = \bot$.
   (b) $S\phi\downarrow \neq \bot$ and is not a pair Then by IH there exists $c' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c'$, which implies that $R\phi'\downarrow$ and is not a pair either by Lemma A.17. Hence $R\phi\downarrow = R\phi'\downarrow = \bot$.
   (c) $S\phi\downarrow = \langle M_1, M_2\rangle$ is a pair Then by IH there exists $c' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c'$. This implies, by Lemma A.17, that $S\phi'\downarrow = \langle M_1', M_2'\rangle$ is also a pair, and that $\Gamma \vdash M_1 \sim M_1' : \mathtt{LL} \to c''$ for some $c'' \subseteq c'$. Since $R\phi\downarrow = M_1$ and $R\phi'\downarrow = M_1'$, this proves the case.
11. $R = \pi_2(S)$ This case is analogous to the case 10.
12. $R = \mathtt{dec}(S, K)$ We apply the induction hypothesis to K and, similarly to the case 6, we distinguish several cases.
   (a) If $K\phi\downarrow = \bot$ or is not a key then, as in case 6, $R\phi\downarrow = R\phi'\downarrow = \bot$.
   (b) If $K\phi\downarrow$ is a key, then similarly to case 6 we can show that $K\phi\downarrow = K\phi'\downarrow$, and either $\Gamma(K\phi\downarrow, K\phi'\downarrow) <: \mathrm{key}^{\mathtt{LL}}(T)$ for some T, or $K\phi\downarrow \in \mathcal{FK}$. We then apply the IH to S, which creates two cases. Either $S\phi\downarrow = S\phi'\downarrow = \bot$, or there exists $c' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c'$. In the first case, the claim holds, since $R\phi\downarrow = R\phi'\downarrow = \bot$. In the second case, by Lemmas A.15 and A.18, and since Γ is well-formed, we know that $S\phi\downarrow$ is a message encrypted with $K\phi\downarrow$ if and only if $S\phi'\downarrow$ also is an encryption by this key. Consequently, if $S\phi\downarrow$ is not an encryption by $K\phi\downarrow$ $(= K\phi'\downarrow)$, then it is the same for $S\phi'\downarrow$; and $R\phi\downarrow = R\phi'\downarrow = \bot$. Otherwise, $S\phi\downarrow = \mathtt{enc}(M, K\phi\downarrow)$ and $S\phi'\downarrow = \mathtt{enc}(N, K\phi'\downarrow)$ for some M, N. In that case, by IH, we have $\Gamma \vdash \mathtt{enc}(M, K\phi\downarrow) \sim \mathtt{enc}(N, K\phi'\downarrow) : \mathtt{LL} \to c'$. Therefore, by Lemma A.15, $\Gamma \vdash M \sim N : \mathtt{LL} \to c'$, which is to say $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c'$. Hence the claim holds in this case.
13. $R = \mathtt{adec}(S, K)$ We apply the induction hypothesis to K and, similarly to the case 6, we distinguish several cases.
   (a) If $K\phi\downarrow = \bot$ or is not a key then, as in case 6, $R\phi\downarrow = R\phi'\downarrow = \bot$.
   (b) If $K\phi\downarrow$ is a key, then similarly to case 6 we can show that $K\phi\downarrow = K\phi'\downarrow$, and either $\Gamma(K\phi\downarrow, K\phi'\downarrow) <: \mathrm{key}^{\mathtt{LL}}(T)$ for some T, or $K\phi\downarrow \in \mathcal{FK}$. We then apply the IH to S, which creates two cases. Either $S\phi\downarrow = S\phi'\downarrow = \bot$, or there exists $c' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c'$. In the first case, the claim holds, since $R\phi\downarrow = R\phi'\downarrow = \bot$. In the second case, by Lemmas A.15 and A.18, and since Γ is well-formed, we know that $S\phi\downarrow$ is a message asymmetrically encrypted by $\mathtt{pk}(K\phi\downarrow)$ if and only if $S\phi'\downarrow$ also is an asymmetric encryption by this key. Consequently, if $S\phi\downarrow$ is not an encryption by $\mathtt{pk}(K\phi\downarrow)$ $(= \mathtt{pk}(K\phi'\downarrow))$, then it is the same for $S\phi'\downarrow$, and $R\phi\downarrow = R\phi'\downarrow = \bot$. Otherwise,

$S\phi\downarrow = \mathtt{aenc}(M, \mathtt{pk}(K\phi\downarrow))$ and $S\phi'\downarrow = \mathtt{aenc}(N, \mathtt{pk}(K\phi'\downarrow))$ for some M, N. Thus by IH we have $\Gamma \vdash \mathtt{aenc}(M, \mathtt{pk}(K\phi\downarrow)) \sim \mathtt{aenc}(N, \mathtt{pk}(K\phi'\downarrow)) : \mathtt{LL} \to c'$. Therefore, by Lemma A.15 (point 6), we know that $\Gamma \vdash M \sim N : \mathtt{LL} \to c'$, which is to say $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c'$. Hence the claim holds in this case.

14. $R = \mathtt{checksign}(S, K)$ We apply the induction hypothesis to K and, similarly to the case 7, we distinguish several cases.

    (a) If $K\phi\downarrow = \bot$ or is not a verification key then, as in case 7, we can show that $R\phi\downarrow = R\phi'\downarrow = \bot$.

    (b) If $K\phi\downarrow$ is a verification key $\mathtt{vk}(k)$ for some $k \in \mathcal{K}$, then similarly to case 7 we can show that $K\phi\downarrow = K\phi'\downarrow$, and $k \in \mathrm{keys}(\Gamma) \cup \mathcal{FK}$. We then apply the IH to S, which creates two cases. Either $S\phi\downarrow = S\phi'\downarrow = \bot$, or there exists $c' \subseteq c$ such that $\Gamma \vdash S\phi\downarrow \sim S\phi'\downarrow : \mathtt{LL} \to c'$. In the first case, the claim holds, since $R\phi\downarrow = R\phi'\downarrow = \bot$. In the second case, by Lemmas A.16 and A.18, and since Γ is well-formed, we know that $S\phi\downarrow$ is a signature by $k (= K\phi\downarrow)$ if and only if $S\phi'\downarrow$ also is a signature by this key. Consequently, if $S\phi\downarrow$ is not signed by k, then neither is $S\phi'\downarrow$, and $R\phi\downarrow = R\phi'\downarrow = \bot$. Otherwise, $S\phi\downarrow = \mathtt{sign}(M, k)$ and $S\phi'\downarrow = \mathtt{sign}(N, k)$ for some M, N. Thus by IH we have $\Gamma \vdash \mathtt{sign}(t, k) \sim \mathtt{sign}(t', k) : \mathtt{LL} \to c'$. Therefore, by Lemma A.16 (point 2), we know that there exists $c'' \subseteq c'$ such that $\Gamma \vdash M \sim N : \mathtt{LL} \to c''$. That is to say $\Gamma \vdash R\phi\downarrow \sim R\phi'\downarrow : \mathtt{LL} \to c''$. Hence the claim holds in this case.

□

Lemma A.22 (Substitution preserves typing)**.** *For all Γ, Γ', such that $\Gamma \cup \Gamma' \vdash \diamond$, (we do not require that Γ' is well-formed), for all M, N, T, c_σ, c, for all ground substitutions σ, σ', if*

- *Γ' only contains variables;*
- *Γ and Γ' have disjoint domains;*
- *for all $x \in \mathrm{dom}(\Gamma)$, $\Gamma(x)$ is not of the form $[\![\tau_m^{l,1}; \tau_n^{l',1}]\!]$,*
- *for all $x \in \mathrm{dom}(\sigma)$, $\sigma(x)\downarrow = \sigma(x)$, and similarly for σ',*
- *$(\Gamma_{\mathcal{N},\mathcal{K}} \cup \Gamma')_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : (\Gamma_{\mathcal{N},\mathcal{K}} \cup \Gamma')_{\mathcal{X}} \to c_\sigma$,*
- *and $\Gamma \cup \Gamma' \vdash M \sim N : T \to c$*

then there exists c' such that $[\![c]\!]_{\sigma,\sigma'} \subseteq c' \subseteq [\![c]\!]_{\sigma,\sigma'} \cup c_\sigma$ and

$$\Gamma \vdash M\sigma \sim N\sigma' : T \to c'.$$

In particular, if we have $\Gamma' = \Gamma'''_{\mathcal{X}}$ and $\Gamma = \Gamma'''_{\mathcal{N},\mathcal{K}}$ for some Γ''', then the first three conditions trivially hold.

Proof. Note that $\Gamma'_{\mathcal{N},\mathcal{K}} = \emptyset$, and $\Gamma'_{\mathcal{X}} = \Gamma'$.

This proof is done by induction on the typing derivation for the terms. The claim clearly holds in the TNonce, TNonceL, TCstFN, TPubKeyL, TVKeyL, TKey, TLR^1, TLR$^\infty$ since their conditions do not use $\Gamma(x)$ (for any variable x) or another type judgement, and they still apply to the messages $M\sigma$ and $N\sigma'$.

In the THIGH case, we have

$$\mathrm{names}(\sigma) \cup \mathrm{names}(\sigma') \cup \mathrm{vars}(\sigma) \cup \mathrm{vars}(\sigma') \cup \mathrm{keys}(\sigma) \cup \mathrm{keys}(\sigma') \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{keys}(\Gamma) \cup \mathcal{FN} \cup \mathcal{FK},$$

and

$$\begin{aligned}&\mathrm{names}(M) \cup \mathrm{names}(N) \cup \mathrm{vars}(M) \cup \mathrm{vars}(N) \cup \mathrm{keys}(M) \cup \mathrm{keys}(N)\\ &\quad\subseteq \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\sigma) \cup \mathrm{keys}(\Gamma) \cup \mathcal{FN} \cup \mathcal{FK},\end{aligned}$$

therefore

$$\begin{aligned}&\mathrm{names}(M\sigma) \cup \mathrm{names}(N\sigma') \cup \mathrm{vars}(M\sigma) \cup \mathrm{vars}(N\sigma') \cup \mathrm{keys}(M\sigma) \cup \mathrm{keys}(N\sigma')\\ &\quad\subseteq \mathrm{dom}(\Gamma) \cup \mathrm{keys}(\Gamma) \cup \mathcal{FN} \cup \mathcal{FK}.\end{aligned}$$

In addition, since $M{\downarrow} \neq \bot$, and $\forall x \in \mathrm{dom}(\sigma), \sigma(x){\downarrow} = \sigma(x)$, we have $M\sigma{\downarrow} \neq \bot$. Similarly, $N\sigma'{\downarrow} \neq \bot$. Therefore, rule THIGH may be applied to obtain $\Gamma \vdash M\sigma \sim N\sigma' : \mathtt{HL} \to \emptyset$.

The claim follows directly from the induction hypothesis in all other cases except the TVAR and TLRVAR cases, which are the base cases.

In the TVAR case, the claim also holds, since $M = N = x$ for some variable $x \in \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma')$. If $x \in \mathrm{dom}(\Gamma)$, then $x\sigma = x\sigma' = x$, and $T = \Gamma(x)$. Thus, by rule TVAR, $\Gamma \cup \Gamma' \vdash x\sigma \sim x\sigma' : \Gamma(x) \to \emptyset$ and the claim holds. If $x \in \mathrm{dom}(\Gamma')$, then $T = \Gamma'(x)$, and, since by hypothesis the substitutions are well-typed, there exists $c_x \subseteq c_{\sigma,\sigma'}$ such that $(\Gamma \cup \Gamma')_{\mathcal{N},\mathcal{K}} \vdash \sigma(x) \sim \sigma'(x) : \Gamma'(x) \to c_x$. Thus, since $(\Gamma \cup \Gamma'')_{\mathcal{N},\mathcal{K}} = \Gamma_{\mathcal{N},\mathcal{K}}$, and by applying Lemma A.10 to Γ, $\Gamma \vdash \sigma(x) \sim \sigma'(x) : \Gamma'(x) \to c_x$ and the claim holds.

Finally, in the TLRVAR case, there exist two variables x, y, and types $\tau_m^{l,1}$, $\tau_n^{l',1}$, $\tau_{m'}^{l'',1}$, $\tau_{n'}^{l''',1}$, such that $M = x$, $N = y$, $c = \emptyset$, $\Gamma \cup \Gamma' \vdash x \sim x : [\![\tau_m^{l,1} ; \tau_n^{l',1}]\!] \to \emptyset$, $\Gamma \cup \Gamma' \vdash y \sim y : [\![\tau_{m'}^{l'',1} ; \tau_{n'}^{l''',1}]\!] \to \emptyset$, and $T = [\![\tau_m^{l,1} ; \tau_{n'}^{l''',1}]\!]$.

By Lemma A.14, this implies that $(\Gamma \cup \Gamma')(x) = [\![\tau_m^{l,1} ; \tau_n^{l',1}]\!]$ and $(\Gamma \cup \Gamma')(y) = [\![\ l'' \ ;\ 1]\!] m' l''' 1 n'$. Hence, since by hypothesis Γ does not contain such types, $x \in \mathrm{dom}(\Gamma')$ and $y \in \mathrm{dom}(\Gamma')$.

Moreover, by the induction hypothesis, there exist c', $c'' \subseteq c_\sigma$ such that $\Gamma \vdash x\sigma \sim x\sigma' : [\![\tau_m^{l,1} ; \tau_n^{l',1}]\!] \to c'$, and $\Gamma \vdash y\sigma \sim y\sigma' : [\![\tau_{m'}^{l'',1} ; \tau_{n'}^{l''',1}]\!] \to c''$. That is to say, since $x, y \in \mathrm{dom}(\Gamma') = \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$, that $\Gamma \vdash \sigma(x) \sim \sigma'(x) : [\![\tau_m^{l,1} ; \tau_n^{l',1}]\!] \to c'$, and $\Gamma \vdash \sigma(y) \sim \sigma'(y) : [\![\tau_{m'}^{l'',1} ; \tau_{n'}^{l''',1}]\!] \to c''$. Hence, by Lemma A.14, and since σ, σ' are ground, we have $\sigma(x) = m$, $\sigma'(x) = n$, $\sigma(y) = m'$, and $\sigma'(y) = n'$, and $\Gamma(m) = \tau_m^{l,1}$ and $\Gamma(n') = \tau_{n'}^{l''',1}$.

Thus, by rule TLR[1], $\Gamma \vdash \sigma(x) \sim \sigma'(y) : [\![\tau_m^{l,1} ; \tau_{n'}^{l''',1}]\!] \to \emptyset$, which proves the claim. □

Lemma A.23 (Types LL and HH are disjoint)**.** *For all well-formed Γ, for all ground terms M, M', N, N', for all sets of constraints c, c', if $\Gamma \vdash M \sim N : \mathtt{LL} \to c$ and $\Gamma \vdash M' \sim N' : \mathtt{HH} \to c'$ then $M \neq M'$ and $N \neq N'$.*

Proof. First, it is easy to see by induction on the type derivation that for all ground terms M, N, for all c, if $\Gamma \vdash M \sim N : \mathtt{HH} \to c$ then either

- M is a nonce $m \in \mathcal{N}$ such that $\Gamma(m) = \tau_m^{\mathtt{HH},a}$ for some $a \in \{\infty, 1\}$;
- or M is a key and there exist $k \in \mathcal{K}$ and T such that $\Gamma(M, k) <: \mathrm{key}^{\mathtt{HH}}(T)$;
- or $\Gamma \vdash M \sim N : \mathtt{HH} * T \to c'$ for some T, c';
- or $\Gamma \vdash M \sim N : T * \mathtt{HH} \to c'$ for some T, c'.

Indeed, (as Γ is well-formed) the only possible cases are TNonce, TSub, and TLR'. In the TNonce case the claim clearly holds. In the TSub case we use Lemma A.1 followed by Lemma A.18. In the TLR' case we apply Lemma A.14 and the claim directly follows.

Let us now show that for all M, N, N' ground, for all c, c', $\Gamma \vdash M \sim N : \mathtt{LL} \to c$ and $\Gamma \vdash M \sim N' : \mathtt{HH} \to c'$ cannot both hold. (This corresponds, with the notations of the statement of the lemma, to proving by contradiction that $M \neq M'$. The proof that $N \neq N'$ is analogous.)

We show this property by induction on the size of M.

Since $\Gamma \vdash M \sim N' : \mathtt{HH} \to c'$, by the property stated in the beginning of this proof, we can distinguish four cases.

- If M is a nonce and $\Gamma(M) = \tau_M^{\mathtt{HH},a}$: then this contradicts Lemma A.18. Indeed, this lemma (point 5) implies that $M \in \mathcal{BN}$, but also (by point 6), since $\Gamma \vdash M \sim N : \mathtt{LL} \to c$, that either $\Gamma(M) = \tau_M^{\mathtt{LL},a}$ for some $a \in \{1, \infty\}$, or $M \in \mathcal{FN} \cup \mathcal{C}$.
- If M is a key and $\Gamma(M, k) <: \mathrm{key}^{\mathtt{HH}}(T)$ for some T, k: then by Lemma A.18, since $\Gamma \vdash M \sim N : \mathtt{LL} \to c$, there exists T', k' such that $\Gamma(M, k') <: \mathrm{key}^{\mathtt{LL}}(T')$ or $k, k' \in \mathcal{FK}$. Since Γ is well-formed, and by Lemma A.1, this contradicts $\Gamma(M) <: \mathrm{key}^{\mathtt{HH}}(T)$.
- If $\Gamma \vdash M \sim N' : \mathtt{HH} * T \to c''$ for some T, c'': then by Lemma A.17, since M, N' are ground, there exist M_1, M_2, N_1', N_2', c_1' such that $M = \langle M_1, M_2 \rangle$, $N' = \langle N_1', N_2' \rangle$, and $\Gamma \vdash M_1 \sim N_1' : \mathtt{HH} \to c_1'$. Moreover, since $\Gamma \vdash M \sim N : \mathtt{LL} \to c$, also by Lemma A.17, there exist N_1, N_2, c_1 such that $N = \langle N_1, N_2 \rangle$ and $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$. However, by the induction hypothesis, $\Gamma \vdash M_1 \sim N_1' : \mathtt{HH} \to c_1'$ and $\Gamma \vdash M_1 \sim N_1 : \mathtt{LL} \to c_1$ is impossible.
- If $\Gamma \vdash M \sim N' : T * \mathtt{HH} \to c''$ for some T, c'': this case is similar to the previous one.

□

Lemma A.24 (Low terms are recipes on their constraints)**.** *For all ground messages M, N, for all $T <: \mathtt{LL}$, for all Γ, c, if $\Gamma \vdash M \sim N : T \to c$ then there exists an attacker recipe R without destructors such that $M = R(\phi_\ell(c) \cup \phi_{\mathtt{LL}}^\Gamma)$ and $N = R(\phi_r(c) \cup \phi_{\mathtt{LL}}^\Gamma)$.*

Proof. We prove this lemma by induction on the typing derivation of $\Gamma \vdash M \sim N : T \to c$. We distinguish several cases for the last rule in this derivation.

- TNonce, THigh, TOr, TLR^1, TLR$^\infty$, TLR', TLRVar: these cases are not possible, since the type they give to terms is never a subtype of $\mathtt{LL}$ by Lemma A.1.
- TVar: this case is not possible since M, N are ground.
- TSub: this case is directly proved by applying the induction hypothesis to the judgement $\Gamma \vdash M \sim N : T' \to c$ where $T' <: T <: \mathtt{LL}$, which appears in the conditions of this rule, and has a shorter derivation.
- TLRL': in this case, $\Gamma \vdash M \sim N : [\![\tau_n^{\mathtt{LL},a} ; \tau_n^{\mathtt{LL},a}]\!] \to c'$ for some nonce n, some $a \in \{\infty, 1\}$, some c', and $c = \emptyset$. By Lemma A.14, this implies that $M = N = n$, and $\Gamma(n) = \tau_n^{\mathtt{LL},a}$. Thus, by definition, there exists x such that $\phi_{\mathtt{LL}}^\Gamma(x) = n$ and the claim holds with $R = x$.

- TNonceL: in this case $M = N = n$ for some $n \in \mathcal{N}$ such that $\Gamma(n) = \tau_n^{\mathtt{LL},a}$ for some $a \in \{1, \infty\}$. Hence, by definition, there exists x such that $\phi^{\Gamma}_{\mathtt{LL}}(x) = n$ and the claim holds with $R = x$.
- TCstFN: then $M = N = a \in \mathcal{C} \cup \mathcal{FN} \cup \mathcal{FK}$, and the claim holds with $R = a$.
- TKey: then by well-formedness of Γ, $M = N = k \in \mathcal{K}$ and there exists T' such that $\Gamma(k, k) <: \mathrm{key}^{\mathtt{LL}}(T')$. By definition, there exists x such that $\phi^{\Gamma}_{\mathtt{LL}}(x) = k$ and the claim holds with $R = x$.
- TPubKeyL, TVKeyL: then $M = N = \mathtt{pk}(k)$ (resp. $\mathtt{vk}(k)$) for some $k \in \mathrm{keys}(\Gamma) \cup \mathcal{FK}$. If $k \in \mathrm{keys}(\Gamma)$, by definition, there exists x such that $\phi^{\Gamma}_{\mathtt{LL}}(x) = \mathtt{pk}(k)$ (resp. $\mathtt{vk}(k)$) and the claim holds with $R = x$. If $k \in \mathcal{FK}$, the claim holds with $R = \mathtt{pk}(k)$.
- TPubKey, TVKey: these two cases are similar, we write the proof for the TPubKey case. The form of this rule application is:

$$\dfrac{\dfrac{\Pi}{\Gamma \vdash M \sim N : T \to \emptyset}}{\Gamma \vdash \mathtt{pk}(M) \sim \mathtt{pk}(N) : \mathrm{pkey}(T) \to \emptyset}$$

  for some T such that $\mathrm{pkey}(T) <: \mathtt{LL}$. By Lemma A.1, this implies that there exist T', l such that $T <: \mathrm{eqkey}^l(T')$. Thus, $\Gamma \vdash M \sim N : \mathrm{eqkey}^l(T') \to \emptyset$. By Lemma A.18, this implies $M = N = k \in \mathrm{keys}(\Gamma)$. By definition, there exists x such that $\phi^{\Gamma}_{\mathtt{LL}}(x) = \mathtt{pk}(k)$, and the claim holds with $R = x$.
- TPair, THashL: these cases are similar. We detail the TPair case. In that case, $T = T_1 * T_2$ for some T_1, T_2. By Lemma A.1, T_1, T_2 are subtypes of $\mathtt{LL}$. In addition, there exist M_1, M_2, N_1, N_2, c_1, c_2 such that $\Gamma \vdash M_i \sim N_i : T_i \to c_i$ (for $i \in \{1, 2\}$). By applying the induction hypothesis to these two judgements (which have shorter proofs), we obtain R_1, R_2 such that for all i, $M_i = R_i(\phi_\ell(c_i) \cup \phi^{\Gamma}_{\mathtt{LL}})$ and $N_i = R_i(\phi_r(c_i) \cup \phi^{\Gamma}_{\mathtt{LL}})$. Therefore the claim holds with $R = \langle R_1, R_2 \rangle$.
- TEncH, TAEncH, THash, TSignH: these four cases are similar. In each case, by the form of the typing rule, we have $c = \{M \sim N\} \cup c'$ for some c'. Therefore by definition of $\phi_\ell(c)$, $\phi_r(c)$, there exists x such that $\phi_\ell(c)(x) = M$ and $\phi_r(c)(N) = N$. The claim holds with $R = x$.
- TEncL, TAEncL: these two cases are similar, we write the proof for the TEncL case. The form of this rule application is:

$$\dfrac{\dfrac{\Pi}{\Gamma \vdash M \sim N : (\mathtt{LL})_T \to c}}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}$$

  for some T such that $T = \mathtt{LL}$ or $T <: \mathrm{key}^{\mathtt{LL}}(T')$ for some T'. In both cases $T <: \mathtt{LL}$. By Lemma A.15, there exist M', N', M'', N'', c', c'' such that $M = \mathtt{enc}(M', M'')$, $N = \mathtt{enc}(N', N'')$, $c = c' \cup c''$, $\Gamma \vdash M' \sim N' : \mathtt{LL} \to c'$ and $\Gamma \vdash M'' \sim N'' : T \to c''$, both with a proof shorter that Π. Thus by applying the induction hypothesis to these judgements, there exist R, R' such that $M' = R(\phi_\ell(c') \cup \phi^{\Gamma}_{\mathtt{LL}})$, $N' = R(\phi_r(c') \cup \phi^{\Gamma}_{\mathtt{LL}})$, $M'' = R'(\phi_\ell(c'') \cup \phi^{\Gamma}_{\mathtt{LL}})$, and $N'' = R'(\phi_r(c'') \cup \phi^{\Gamma}_{\mathtt{LL}})$.

  Therefore, the claim holds with the recipe $\mathtt{enc}(R, R')$.

- TEnc, TAenc: these two cases are similar, we write the proof for the TEnc case. The form of this rule application is:

$$\frac{\Pi}{\Gamma \vdash M \sim N : T \to c} \qquad \frac{\Pi'}{\Gamma \vdash M' \sim N' : T' \to c'}$$
$$\Gamma \vdash \mathtt{enc}(M, M') \sim \mathtt{enc}(N, N') : (T)_{T'} \to c \cup c'$$

  for some T, T' such that $(T)_{T'} <: \mathtt{LL}$. By Lemma A.1, $T <: \mathtt{LL}$ and $T' <: \mathtt{LL}$. We conclude the proof of this case similarly to the TEncL case.

- TSignL: the form of this rule application is:

$$\frac{\Pi}{\Gamma \vdash M' \sim N' : \mathtt{LL} \to c} \qquad \frac{\Pi}{\Gamma \vdash M'' \sim N'' : \mathtt{LL} \to c'}$$
$$\Gamma \vdash \mathtt{sign}(M', M'') \sim \mathtt{sign}(N', M'') : \mathtt{LL} \to c \cup c'$$

  with $M = \mathtt{sign}(M', M'')$, $N = \mathtt{sign}(N', N'')$. Thus by applying the induction hypothesis to the two hypotheses of the rule, *i.e.* $\Gamma \vdash M' \sim N' : \mathtt{LL} \to c$ and $\Gamma \vdash M'' \sim N'' : \mathtt{LL} \to c'$ there exist R, R' such that $M' = R(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}})$, $N' = R(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$, $M'' = R'(\phi_\ell(c') \cup \phi^\Gamma_{\mathtt{LL}})$, and $N'' = R'(\phi_r(c') \cup \phi^\Gamma_{\mathtt{LL}})$.

  Therefore, the claim holds with the recipe $\mathtt{sign}(R, R')$.

□

Lemma A.25 (Low frames with consistent constraints are statically equivalent)**.** *For all ground ϕ, ϕ', for all c, Γ, if*

- $\Gamma \vdash \phi \sim \phi' : \mathtt{LL} \to c$
- *and c is consistent in $\Gamma_{\mathcal{N},\mathcal{K}}$,*

then ϕ and ϕ' are statically equivalent.

Proof. We can first notice that since ϕ and ϕ' are ground, so is c (this is easy to see by examining the typing rules for terms). Let R, R' be two attacker recipes, such that $\mathrm{vars}(R) \cup \mathrm{vars}(R') \subseteq \mathrm{dom}(\phi) (= \mathrm{dom}(\phi'))$.

For all $x \in \mathrm{dom}(\phi) (= \mathrm{dom}(\phi'))$, by assumption, there exists $c_x \subseteq c$ such that $\Gamma \vdash \phi(x) \sim \phi'(x) : \mathtt{LL} \to c_x$. By Lemma A.24, there exists a recipe R_x such that $\phi(x) = R_x(\phi_\ell(c_x) \cup \phi^\Gamma_{\mathtt{LL}})$ and $\phi'(x) = R_x(\phi_r(c_x) \cup \phi^\Gamma_{\mathtt{LL}})$.

Since $c_x \subseteq c$, we also have $\phi(x) = R_x(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}})$ and $\phi'(x) = R_x(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$.

Let $\overline{R}$ and $\overline{R}'$ be the recipes obtained by replacing every occurence of x with R_x in respectively R and R', for all variable $x \in \mathrm{dom}(\phi) (= \mathrm{dom}(\phi'))$.

We then have $R\phi = \overline{R}(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}})$ and $R'\phi = \overline{R}'(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}})$; and similarly $R\phi' = \overline{R}(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$ and $R'\phi' = \overline{R}'(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$.

Since c is ground, and consistent in $\Gamma_{\mathcal{N},\mathcal{K}}$, by definition of consistency, the frames $\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}}$ and $\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}}$ are statically equivalent. Hence, by definition of static equivalence,

$$\overline{R}(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}}) = \overline{R}'(\phi_\ell(c) \cup \phi^\Gamma_{\mathtt{LL}}) \iff \overline{R}(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}}) = \overline{R}'(\phi_r(c) \cup \phi^\Gamma_{\mathtt{LL}})$$

i.e.

$$R\phi = R'\phi \iff R\phi' = R'\phi'$$

Therefore, ϕ and $.\phi'$ are statically equivalent. □

Lemma A.26 (Invariant)**.** *For all* Γ, ϕ_P, ϕ'_P, ϕ_Q, σ^1_P, σ^2_P, σ^1_Q,c_ϕ, c_σ, *for all multisets of processes* $\mathcal{P}$, $\mathcal{P}'$, $\mathcal{Q}$, *where the processes in* $\mathcal{P}$, $\mathcal{P}'$, $\mathcal{Q}$ *are noted* $\{P_i\}$, $\{P'_i\}$, $\{Q_i\}$*; for all constraint sets* $\{C_i\}$, *if:*

- $|\mathcal{P}| = |\mathcal{Q}|$
- $\mathrm{dom}(\phi_P) = \mathrm{dom}(\phi_Q)$
- $\forall i$, *there is a derivation* Π_i *of* $\Gamma \vdash P_i \sim Q_i \to C_i$,
- $\Gamma \vdash \phi_P \sim \phi_Q : \mathtt{LL} \to c_\phi$
- *for all* $i \neq j$, *the sets of bound variables in* P_i *and* P_j *(resp.* Q_i *and* Q_j*) are disjoint, and similarly for the names*
- σ^1_P, σ^1_Q *are ground, and there exist ground* $\sigma_P \supseteq \sigma^1_P$, $\sigma_Q \supseteq \sigma^1_Q$ *such that*
  - $(\mathrm{dom}(\sigma_P) \backslash \mathrm{dom}(\sigma^1_P)) \cap (\mathrm{vars}(\mathcal{P}) \cup \mathrm{vars}(\phi_P)) = \emptyset$,
  - $(\mathrm{dom}(\sigma_Q) \backslash \mathrm{dom}(\sigma^1_Q)) \cap (\mathrm{vars}(\mathcal{Q}) \cup \mathrm{vars}(\phi_Q)) = \emptyset$, *and*
  - $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$,
  - *for all* $x \in \mathrm{dom}(\sigma_P)$, $\sigma_P(x)\!\downarrow = \sigma_P(x)$, *and similarly for* σ_Q,
- $c_\sigma \subseteq [\![c_\phi]\!]_{\sigma^1_P,\sigma^1_Q}$,
- $[\![(\cup_{\times i} C_i) \cup_\forall c_\phi]\!]_{\sigma^1_P,\sigma^1_Q}$ *is consistent,*
- $(\mathcal{P}, \phi_P, \sigma^1_P) \xrightarrow{\alpha} (\mathcal{P}', \phi'_P, \sigma^2_P)$,

then there exist a word w, *a multiset* $\mathcal{Q}' = \{Q'_i\}$, *constraint sets* $\{C'_i\}$, *a frame* ϕ'_Q, *a substitution* σ'_Q, *an environment* Γ', *constraints* c'_ϕ, *and* c'_σ *such that:*

- $w =_\tau \alpha$
- $|\mathcal{P}'| = |\mathcal{Q}'|$
- *for all* $i \neq j$, *the sets of bound variables in* P'_i *and* P'_j *(resp.* Q'_i *and* Q'_j*) are disjoint, and similarly for the bound names;*
- $\mathrm{fvars}(\mathcal{P}') \cup \mathrm{fvars}(\phi'_P) \subseteq \mathrm{dom}(\sigma^2_P)$ *and* $\mathrm{fvars}(\mathcal{Q}') \cup \mathrm{fvars}(\phi'_Q) \subseteq \mathrm{dom}(\sigma^2_Q)$
- $\Gamma' \vdash \phi'_P \sim \phi'_Q : \mathtt{LL} \to c'_\phi$
- $(\mathcal{Q}, \phi_Q, \sigma^1_Q) \xrightarrow{w}_* (\mathcal{Q}', \phi'_Q, \sigma^2_Q)$,
- $\forall i, \Gamma' \vdash P'_i \sim Q'_i \to C'_i$,
- σ^2_P, σ^2_Q *are ground and there exist* $\sigma'_P \supseteq \sigma^2_P$, *and* $\sigma'_Q \supseteq \sigma^2_Q$, *such that*
  - $(\mathrm{dom}(\sigma'_P) \backslash \mathrm{dom}(\sigma^2_P)) \cap (\mathrm{vars}(\mathcal{P}') \cup \mathrm{vars}(\phi'_P)) = \emptyset$,
  - $(\mathrm{dom}(\sigma'_Q) \backslash \mathrm{dom}(\sigma^2_Q)) \cap (\mathrm{vars}(\mathcal{Q}') \cup \mathrm{vars}(\phi'_Q)) = \emptyset$, *and*
  - $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma'_P \sim \sigma'_Q : \Gamma'_{\mathcal{X}} \to c'_\sigma$,
  - *for all* $x \in \mathrm{dom}(\sigma'_P)$, $\sigma'_P(x)\!\downarrow = \sigma'_P(x)$, *and similarly for* σ'_Q,

- $\mathrm{dom}(\phi'_P) = \mathrm{dom}(\phi'_Q)$,
- $c'_\sigma \subseteq \left[\!\left[c'_\phi\right]\!\right]_{\sigma_P^2,\sigma_Q^2}$,
- $\left[\!\left[(\cup_{\times_i} C'_i)\cup_\forall c'_\phi\right]\!\right]_{\sigma_P^2,\sigma_Q^2}$ *is consistent.*

Proof. Note that the assumption that σ_P (resp. σ_Q) extends σ_P^1 (resp. σ_Q^1) only with variables not appearing in $\mathcal{P}$ or ϕ_P (resp. $\mathcal{Q}$ or ϕ_Q) implies that $[\![(\cup_{\times_i} C_i)\cup_\forall c_\phi]\!]_{\sigma_P^1,\sigma_Q^1} = [\![(\cup_{\times_i} C_i)\cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$. Similarly, we have $\left[\!\left[(\cup_{\times_i} C'_i)\cup_\forall c'_\phi\right]\!\right]_{\sigma_P^2,\sigma_Q^2} = \left[\!\left[(\cup_{\times_i} C'_i)\cup_\forall c'_\phi\right]\!\right]_{\sigma'_P,\sigma'_Q}$, $[\![c_\phi]\!]_{\sigma_P^1,\sigma_Q^1} = [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$, and $\left[\!\left[c'_\phi\right]\!\right]_{\sigma_P^2,\sigma_Q^2} = \left[\!\left[c'_\phi\right]\!\right]_{\sigma'_P,\sigma'_Q}$.

First, we show that it is sufficient to prove this lemma in the case where Γ does not contain any union types. Indeed, assume we know the property holds in that case. Let us show that the lemma then also holds in the other case, *i.e.* if Γ contains union types. By hypothesis, σ_P, σ_Q are ground, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_\mathcal{X} \to c_\sigma$. Hence we know by Lemma A.4 that there exists a branch $\Gamma'' \in \mathrm{branches}(\Gamma)$ (thus Γ'' does not contain union types), such that $(\Gamma'')_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : (\Gamma'')_\mathcal{X} \to c_\sigma$.

Moreover, by Lemma A.7, $\forall i, \Gamma'' \vdash P_i \sim Q_i \to C''_i \subseteq C_i$; and by Lemma A.5, $\Gamma'' \vdash \phi_P \sim \phi_Q : \mathtt{LL} \to c_\phi$. In addition by Lemma A.11, $[\![(\cup_{\times_i} C''_i)\cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is a subset of $[\![(\cup_{\times_i} C_i)\cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ and is therefore consistent. Thus, if the lemma holds when the environment does not contain union types, it can be applied to the same processes, frames, substitutions and to Γ'', which directly concludes the proof.

Therefore, we may assume that Γ does not contain any union types.

Note that, since by assumption $c_\sigma \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$, we have

$$[\![(\cup_{\times_i} C_i)\cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} = \left[\!\left[(\cup_{\times_i} C''_i)\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q} \cup_\forall c_\sigma.$$

Hence, by Lemma A.11, $(\cup_{\times_i} [\![C_i]\!]_{\sigma_P,\sigma_Q} \cup_\forall c_\sigma)$ is consistent. Thus the assumption on the disjointness of the sets of bound variables (and names) in the processes implies, using Lemma A.8, that each of the $[\![C_i]\!]_{\sigma_P,\sigma_Q} \cup_\forall c_\sigma$ is also consistent. Moreover, this disjointness property for $\mathcal{P}'$ and $\mathcal{Q}'$ follows from the other points, as it is easily proved by examining the reduction rules that it is preserved by reduction.

By hypothesis, $(\mathcal{P}, \phi_P, \sigma_P^1)$ reduces to $(\mathcal{P}', \phi'_P, \sigma_P^2)$ We know from the form of the reduction rules that exactly one process $P_i \in \mathcal{P}$ is reduced, while the others are unchanged. By the assumptions, there is a corresponding process $Q_i \in \mathcal{Q}$ and a derivation Π_i of $\Gamma \vdash P_i \sim Q_i \to C_i$.

We continue the proof by a case disjunction on the last rule of Π_i. Let us first consider the cases of the rules PZero, PPar, PNew, and POr.

- PZero: then $P_i = Q_i = 0$. Hence, the reduction rule applied to $\mathcal{P}$ is Zero, and $\mathcal{P}' = \mathcal{P}\backslash\{P_i\}$, $\phi'_P = \phi_P$, and $\sigma_P^2 = \sigma_P^1$. The same reduction can be performed in $\mathcal{Q}$:

  $$(\mathcal{Q}, \phi_Q, \sigma_Q^1) \xrightarrow{\tau} (\mathcal{Q}\backslash\{Q_i\}, \phi_Q, \sigma_Q^1)$$

  Since the other processes, the frames, environments and substitutions do not change in this reduction, all the claims clearly hold in this case (with $\sigma'_P = \sigma_P$, $\sigma'_Q = \sigma_Q$, $c'_\phi = c_\phi$,

$c'_\sigma = c_\sigma$). In particular, the consistency of the constraints follows from the consistency hypothesis. Indeed,

$$\begin{aligned}(\cup_{\times j\neq i} C_j)\cup_\times C_i \cup_\forall c_\phi &= (\cup_{\times j\neq i} C_j)\cup_\times \{(\emptyset,\Gamma)\}\cup_\forall c_\phi \\ &= (\cup_{\times j\neq i} C_j)\cup_\forall c_\phi,\end{aligned}$$

since Γ is already contained in the environments appearing in each C_j (by Lemma A.12). Thus

$$[\![(\cup_{\times j} C'_j)\cup_\forall c'_\phi]\!]_{\sigma'_P,\sigma'_Q} = [\![(\cup_{\times j} C_j)\cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$$

- PPar: then $P_i = P_i^1 \mid P_i^2$, $Q_i = Q_i^1 \mid Q_i^2$. Hence, the reduction rule applied to $\mathcal{P}$ is Par:

  $$(\mathcal{P}, \phi_P, \sigma_P^1) \xrightarrow{\tau} (\mathcal{P}\backslash\{P_i\}\cup\{P_i^1, P_i^2\}, \phi_P, \sigma_P^1).$$

  We choose $\Gamma' = \Gamma$.

  In addition

  $$\Pi_i = \frac{\dfrac{\Pi^1}{\Gamma \vdash P_i^1 \sim Q_i^1 \to C_i^1} \qquad \dfrac{\Pi^2}{\Gamma \vdash P_i^2 \sim Q_i^2 \to C_i^2}}{\Gamma \vdash P_i \sim Q_i \to C_i = C_i^1 \cup_\times C_i^2}.$$

  The same reduction rule can be applied to $\mathcal{Q}$:

  $$(\mathcal{Q}, \phi_Q, \sigma_Q^1) \xrightarrow{\tau} (\mathcal{Q}\backslash\{Q_i\}\cup\{Q_i^1, Q_i^2\}, \phi_Q, \sigma_Q^1)$$

  In this case again, the claims on the substitutions and frames, as well as the claim that $c'_\sigma \subseteq [\![c'_\phi]\!]_{\sigma_P^2,\sigma_Q^2}$, hold since they do not change in the reduction. Moreover the processes in $\mathcal{P}'$ and $\mathcal{Q}'$ are still pairwise typably equivalent. Indeed, all the processes from $\mathcal{P}$ and $\mathcal{Q}$ are unchanged, except for P_i and Q_i which are reduced to P_i^1, P_i^2, Q_i^1, Q_i^2, and those are typably equivalent using Π^1 and Π^2.

  Finally the constraint set is still consistent, since:

  $$\begin{aligned}[\![(\cup_{\times j} C'_j)\cup_\forall c'_\phi]\!]_{\sigma'_P,\sigma'_Q} &= [\![(\cup_{\times j\neq i} C_j)\cup_\times C_i^1 \cup_\times C_i^2 \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} \\ &= [\![(\cup_{\times j} C_j)\cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}\end{aligned}$$

- PNew: then $P_i = \texttt{new}\ n : \tau_n^{l,a}.P'_i$ and $Q_i = \texttt{new}\ n : \tau_n^{l,a}.Q'_i$. P_i is reduced to P'_i by rule New:

  $$(\mathcal{P}, \phi_P, \sigma_P^1) \xrightarrow{\tau} (\mathcal{P}\backslash\{P_i\}\cup\{P'_i\}, \phi_P, \sigma_P^1).$$

  In addition

  $$\Pi_i = \frac{\dfrac{\Pi'_i}{\Gamma, n : \tau_n^{l,a} \vdash P'_i \sim Q'_i \to C_i}}{\Gamma \vdash P_i \sim Q_i \to C_i}.$$

  We choose $\Gamma' = \Gamma, n : \tau_n^{l,a}$.

  The same reduction rule can be applied to $\mathcal{Q}$:

  $$(\mathcal{Q}, \phi_Q, \sigma_Q^1) \xrightarrow{\tau} (\mathcal{Q}\backslash\{Q_i\}\cup\{Q'_i\}, \phi_Q, \sigma_Q^1)$$

  The claim clearly holds. Indeed the processes are still pairwise typable:

  - using Π'_i in the case of P'_i and Q'_i;
  - using Π_j, for $j \neq i$, as well as Lemma A.10, for the other processes, since n does not appear in these processes by assumption.

  In addition, all the frames, substitutions, and constraints are unchanged; and σ, σ' are well-typed in Γ' if and only if they are well-typed in Γ.

- PNewKey: This case is analogous to the PNew case.

- POr: this case is not possible, since we have already eliminated the case where Γ contains union types.

In all the other cases for the last rule in Π_i, we know that the head symbol of P_i is not $\mid$, 0 or `new`.

Hence, the form of the reduction rules implies that $P_i \in \mathcal{P}$ is reduced to exactly one process $P'_i \in \mathcal{P}'$, while the other processes in $\mathcal{P}$ do not change (*i.e.* $P'_j = P_j$ for $j \neq i$). If we show in each case that the same reduction rule that is applied to P_i can be applied to reduce $\mathcal{Q}$ to a multiset $\mathcal{Q}'$ by reducing process Q_i into Q'_i, we will also have $Q'_j = Q_j$ for $j \neq i$. Therefore the claim on the cardinality of the processes multisets will hold.

Since P_i, Q_i can be typed and the head symbol of P_i is not `new`, it is clear by examining the typing rules that the head symbol of Q_i is not `new` either. Hence, we will choose a Γ' containing the same nonces and keys as Γ.

The proofs for theses cases follow the same structure:

- The typing rule gives us information on the form of P_i and Q_i.
- The form of P_i gives us information on which reduction rule was applied to $\mathcal{P}$.
- The form of Q_i is the same as P_i. Hence (additional conditions may need to be checked depending on the rule) Q_i can be reduced to some process Q'_i by applying the same reduction rule that was applied to P_i (or at least, a reduction rule with the same action).
- thus $\mathcal{Q}$ can be reduced too, with the same actions as $\mathcal{P}$. We then check the additional conditions on the typing of the processes, frames and substitutions, and the consistency condition.

First, let us consider the POut case.

- POut: then $P_i = \mathtt{out}(M).P'_i$ and reduces to P'_i via the Out rule, and $Q_i = \mathtt{out}(N).Q'_i$ for some N and Q'_i. Since the Out rule can be applied to P_i, $M\sigma^1_P{\downarrow} \neq \bot$, *i.e.* $M\sigma^1_P{\downarrow} = M\sigma^1_P$. In addition

$$\Pi_i = \dfrac{\dfrac{\Pi}{\Gamma \vdash P'_i \sim Q'_i \to C'_i} \qquad \dfrac{\Pi'}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}}{\Gamma \vdash P_i \sim Q_i \to C_i = C'_i \cup_\forall c}.$$

  We have $\sigma^2_P = \sigma^1_P$, $\phi'_P = \phi_P \cup \{M/ax_n\}$, and $\alpha = \mathtt{new}\ ax_n.\mathtt{out}(ax_n)$.

  Since $\Gamma \vdash M \sim N : \mathtt{LL} \to c$ and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$, by Lemma A.22, we have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma_P \sim N\sigma_Q : \mathtt{LL} \to c'$ for some c'. That is to say $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma^1_P \sim N\sigma^1_Q : \mathtt{LL} \to c'$. Since we also know that $M\sigma^1_P{\downarrow} \neq \bot$, by Lemma A.19, we also have $N\sigma^1_Q{\downarrow} \neq \bot$.

Hence, the same reduction rule Out can be applied to reduce the process Q_i into Q'_i, and the claim on the reduction of $\mathcal{Q}$ holds. We choose $\Gamma' = \Gamma$. We have $\sigma^2_Q = \sigma^1_Q$, and $\phi'_Q = \phi_Q \cup \{N/ax_n\}$. We also choose $\sigma'_P = \sigma_P$, $\sigma'_Q = \sigma_Q$, $c'_\phi = c_\phi \cup c$ and $c'_\sigma = c_\sigma$. The substitutions σ^1_P, σ^1_Q are not extended by the reduction, and the typing environment does not change, which trivially proves the claim regarding the substitutions.

In addition, since by assumption $c_\sigma \subseteq [\![c_\phi]\!]_{\sigma^1_P,\sigma^1_Q}$, and since $c_\phi \subseteq c'_\phi$, we have $c'_\sigma \subseteq \left[\!\!\left[c'_\phi\right]\!\!\right]_{\sigma^2_P,\sigma^2_Q}$.

Moreover, since only M and N are added to the frames in the reduction, Π' suffices to prove the claim that $\Gamma \vdash \phi'_P \sim \phi'_Q : \mathtt{LL} \to c'_\phi$. Since all processes other that P_i and Q_i are unchanged by the reduction (and since the typing environment is also unchanged), Π suffices to proves the claim that $\forall j.\, \Gamma' \vdash P'_j \sim Q'_j \to C'_j$ (with $C'_j = C_j$ for $j \neq i$).

Thus, in this case, it only remains to be proved that $\left[\!\!\left[(\cup_{\times j} C'_j) \cup_\forall c'_\phi\right]\!\!\right]_{\sigma'_P,\sigma'_Q}$ is consistent.

This constraint set is equal to

$$[\![(\cup_{\times j\neq i} C_j) \cup_\times C'_i \cup_\forall (c_\phi \cup c)]\!]_{\sigma_P,\sigma_Q}$$

i.e. to

$$[\![(\cup_{\times j\neq i} C_j) \cup_\times (C'_i \cup_\forall c) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$$

i.e.

$$[\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$$

which is consistent by hypothesis. Hence the claim holds in this case.

In the remaining cases, from the form of the typing rules for processes, the head symbol of neither P_i nor Q_i is `out`. Thus, the reduction applied to P_i (from the assumption), as well as the one applied to Q_i (which, as we will show, has the same action as the rule for P_i), cannot be Out. Therefore no new term is output on either side, and $\phi'_P = \phi_P$ and $\phi'_Q = \phi_Q$. Hence the claim on the domains of the frames holds by assumption. Moreover, as we will see, in all cases Γ' is either Γ, or $\Gamma, x : T$ where x is a variable bound in (the head of) P_i and Q_i, and T is not a union type.

We choose $c'_\phi = c_\phi$. The claim that $\Gamma' \vdash \phi'_P \sim \phi'_Q : \mathtt{LL} \to c'_\phi$ is then actually that $\Gamma' \vdash \phi_P \sim \phi_Q : \mathtt{LL} \to c_\phi$, which is true by Lemma A.10, since by hypothesis $\Gamma \vdash \phi_P \sim \phi_Q : \mathtt{LL} \to c_\phi$.

Besides, in the cases where we choose $\Gamma' = \Gamma$ then it is true (by hypothesis) that for $j \neq i$, $\Gamma' \vdash P'_j \sim Q'_j \to C_j$. In the cases where we choose $\Gamma' = \Gamma, x : T$, where x is bound in P_i and Q_i, then, since the processes are assumed to use different variable names, x does not appear in P_j or Q_j (for $j \neq i$). Hence, if $j \neq i$, using the assumption that $\Gamma \vdash P_j \sim Q_j \to C_j$, by Lemma A.10, we have $\Gamma' \vdash P'_j \sim Q'_j \to C'_j$, where $C'_j = \{(c, \Gamma_c \cup \{x : T\}) | (c, \Gamma_c) \in C_j\}$.

Hence, for each remaining possible last rule of Π_i, we only have to show that:

a) The same reduction rule can be applied to Q_i as to P_i, with the same action. (Except in the case of the rule PIfLR, as we will see, where rule If-Then may be applied on one side while rule If-Else is applied on the other side, but this has no influence on the argument, as these two rules both represent a silent action, and have a very similar form.)

b) there exist σ'_P and σ'_Q ground, and containing σ^2_P and σ^2_Q respectively, that satisfy the conditions on the domains, contain only messages that do not reduce to $\bot$, and such that $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma'_P \sim \sigma'_Q : \Gamma'_\mathcal{X} \to c'_\sigma$ for some set of constraints c'_σ. Since at most one variable x is added to the substitutions in the reduction, we will show in each case that we can choose

these substitutions such that either $\sigma'_P = \sigma_P$ and $\sigma'_Q = \sigma_Q$; or $\sigma'_P = \sigma_P \cup \{M/x\}$ and $\sigma'_Q = \sigma_Q \cup \{N/x\}$ for some messages M, N. In all cases, it is clear from the reduction rules that $M{\downarrow} \neq \bot$ and $N{\downarrow} \neq \bot$. We will then only need to check the well-typedness condition on variable x, *i.e.* $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma'_P(x) \sim \sigma'_Q(x) : \Gamma'(x) \to c_x$ for some c_x. We can then choose $c'_\sigma = c_\sigma \cup c_x$. As we will see in the proof, we will always have $c_x \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q} \cup c_\sigma$.

In addition, $c'_\phi = c_\phi$, and by assumption, x cannot appear in c_ϕ, thus $\left[\!\left[c'_\phi\right]\!\right]_{\sigma'_P,\sigma'_Q} = [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$. Therefore, since by assumption $c_\sigma \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$, the claim that $c'_\sigma \subseteq \left[\!\left[c'_\phi\right]\!\right]_{\sigma'_P,\sigma'_Q}$ will always hold.

c) the new processes obtained by reducing P_i and Q_i are typably equivalent in Γ', with a constraint C'_i, such that

$$\left[\!\left[(\cup_{\times j\neq i} C_j)\cup_\times C'_i \cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q}$$

is consistent.

The actual claim, from the statement of the lemma, is that

$$\left[\!\left[(\cup_{\times j\neq i} C'_j)\cup_\times C'_i \cup_\forall c_\phi\right]\!\right]_{\sigma'_P,\sigma'_Q}$$

is consistent, but we can show that the previous condition is sufficient.

In the case where $\Gamma = \Gamma'$, we have $\sigma'_P = \sigma_P$, $\sigma'_Q = \sigma_Q$, $C'_j = C_j$ for $j \neq i$, and $c'_\sigma = c_\sigma$. Thus the proposed condition is clearly sufficient (it is even necessary in this case).

In the case where $\Gamma' = \Gamma, x : T$ for some T which is not a union type, and the substitutions σ'_P, σ'_Q are σ_P, σ_Q extended with a term associated to x, the proof that the condition is sufficient is more involved. First, we show that $(\cup_{\times j\neq i} C'_j)\cup_\times C'_i = (\cup_{\times j\neq i} C_j)\cup_\times C'_i$. Indeed, if S denotes the set $(\cup_{\times j\neq i} C'_j)\cup_\times C'_i$, we have

$$\begin{aligned}
S &= \{(\bigcup_j c'_j, \bigcup_j \Gamma'_j) \mid \forall j.\ (c'_j, \Gamma'_j) \in C'_j \ \wedge\ \forall j, j'.\ \Gamma'_j \text{ and } \Gamma'_{j'} \text{ are compatible})\} \\
&= \{(c'_i \cup \bigcup_{j\neq i} c_j, \Gamma'_i \cup \bigcup_{j\neq i} (\Gamma_j, x : T) \mid (c'_i, \Gamma'_i) \in C'_i \ \wedge\ (\forall j \neq i.\ (c_j, \Gamma_j) \in C_j) \wedge \\
&\qquad (\forall j \neq i, j' \neq i.\ (\Gamma_j, x : T) \text{ and } (\Gamma_{j'}, x : T) \text{ are compatible}) \wedge \\
&\qquad (\forall j \neq i.\ \Gamma'_i \text{ and } (\Gamma_j, x : T) \text{ are compatible}))\}
\end{aligned}$$

since we already know that for $j \neq i$, $C'_j = \{(c, \Gamma_c \cup \{x : T\}) | (c, \Gamma_c) \in C_j\}$. Assuming we show that $\Gamma, x : T \vdash P'_i \sim Q'_i \to C'_i$, by Lemma A.12, we will also have that all the Γ'_i appearing in the elements of C'_i contain $x : T$ (since T is not a union type). Hence:

$$\begin{aligned}
S = \{&(c'_i \cup \bigcup_{j\neq i} c_j, \Gamma'_i \cup (\bigcup_{j\neq i} \Gamma_j) \mid (c'_i, \Gamma'_i) \in C'_i \ \wedge\ (\forall j \neq i.\ (c_j, \Gamma_j) \in C_j) \wedge \\
&(\forall j \neq i, j' \neq i.\ \Gamma_j \text{ and } \Gamma_{j'} \text{ are compatible}) \ \wedge\ (\forall j \neq i.\ \Gamma'_i \text{ and } \Gamma_j \text{ are compatible}))\} \\
= {} &(\cup_{\times j\neq i} C_j)\cup_\times C'_i
\end{aligned}$$

It is thus sufficient to ensure the consistency of $[\![S \cup_\forall c_\phi]\!]_{\sigma'_P,\sigma'_Q}$. Since $\sigma'_P = \sigma_P \cup \{\sigma'_P(x)/x\}$ (and similarly for Q), it then suffices to show the consistency of $\left[\!\left[[\![S \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}\right]\!\right]_{\sigma'_P(x)/x,\sigma'_Q(x)/x}$.

The substitutions $\sigma'_P(x)$ and $\sigma'_Q(x)$ are ground, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma'_P(x) \sim \sigma'_Q(x) : T \to c_x$ (which we will show for each case as point b)). Hence by Lemma A.11, if $[\![S \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} \cup_\forall c_x$ is consistent, then $\left[\!\left[[\![S \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}\right]\!\right]_{\sigma'_P(x)/x,\sigma'_Q(x)/x}$ is consistent. Moreover, as explained in point b), we will show in each case that $c_x \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$. Thus $[\![S \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} \cup_\forall c_x = [\![S \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$.

Therefore, by the previous implication, it is sufficient to prove that $[\![S \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is consistent, to ensure that $\left[\!\left[[\![S \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}\right]\!\right]_{\sigma'_P(x)/x,\sigma'_Q(x)/x}$ is consistent. This is the condition stated at the beginning of this point, since $S = (\cup_{\times j \neq i} C_j) \cup_\times C'_i$.

We can now prove the remaining cases for the last rule of Π_i, that is to say the cases of the rules PIn, PLet, PLetDec, PLetAdecSame, PLetAdecDiff, PLetLR, PIfL, PIfLR, PIfS, PIfLR*, PIfP, PIfI, PIfLR'*, and PIfAll.

- PIn: then $P_i = \mathtt{in}(x).P'_i$ and reduces to P'_i via the In rule, and $Q_i = \mathtt{in}(x).Q'_i$ for some Q'_i. In addition

$$\Pi_i = \frac{\dfrac{\Pi}{\Gamma, x : \mathtt{LL} \vdash P'_i \sim Q'_i \to C'_i}}{\Gamma \vdash P_i \sim Q_i \to C_i = C'_i}.$$

  We have $\alpha = \mathtt{in}(R)$ for some attacker recipe R such that $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi_P)$, and $R\phi_P\sigma^1_P{\downarrow} = R\phi_P\sigma_P{\downarrow} \neq \bot$. We also have $\sigma^2_P = \sigma^1_P \cup \{R\phi_P\sigma^1_P{\downarrow}/x\}$, $\phi'_P = \phi_P$.

  The same reduction rule In can be applied to reduce the process Q_i into Q'. Indeed,

  - $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi_Q)$ since $\mathrm{dom}(\phi_Q) = \mathrm{dom}(\phi_P)$ by hypothesis;
  - $R\phi_Q\sigma^1_Q{\downarrow} = R\phi_Q\sigma_Q{\downarrow} \neq \bot$. This follows from Lemma A.21, using the fact that by Lemma A.22, $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \phi_P\sigma_P \sim \phi_Q\sigma_Q : \mathtt{LL} \to c$, for some $c \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q} \cup c_\sigma$.

  Therefore point a) holds.

  We choose $\Gamma' = \Gamma, x : \mathtt{LL}$. We have $\sigma^2_Q = \sigma^1_Q \cup \{R\phi_Q\sigma^1_Q{\downarrow}/x\}$. We choose $\sigma'_P = \sigma_P \cup \{R\phi_P\sigma^1_P{\downarrow}/x\}$ and $\sigma'_Q = \sigma_Q \cup \{R\phi_Q\sigma^1_Q{\downarrow}/x\}$.

  Lemmas A.22 and A.21, previously evoked, guarantee that

$$\Gamma_{\mathcal{N},\mathcal{K}} \vdash R\phi_P\sigma_P{\downarrow} \sim R\phi_Q\sigma_Q{\downarrow} : \mathtt{LL} \to c'$$

  for some $c' \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q} \cup c_\sigma$. This proves point b).

  Moreover, Π and the fact that

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C'_i \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} = [\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$$

  which is consistent by hypothesis, prove point c) and conclude this case.

- PLET: then $P_i = \texttt{let}\ x = t\ \texttt{in}\ P_i'\ \texttt{else}\ P_i''$ and $Q_i = \texttt{let}\ x = t'\ \texttt{in}\ Q_i'\ \texttt{else}\ Q_i''$ for some P_i', P_i'', Q_i', Q_i'', t, t'. P_i reduces to either P_i' via the Let-In rule, or P_i'' via the Let-Else rule. In addition

$$\Pi_i = \frac{x \notin \mathrm{dom}(\Gamma) \qquad \dfrac{\Pi}{\Gamma \vdash t \sim t' : T} \qquad \dfrac{\Pi'}{\Gamma, x : T \vdash P_i' \sim Q_i' \to C_i'} \qquad \dfrac{\Pi''}{\Gamma \vdash P_i'' \sim Q_i'' \to C_i''}}{\Gamma \vdash P_i \sim Q_i \to C_i = C_i' \cup C_i''}.$$

  We have $\alpha = \tau$.

  Let $\sigma = \sigma_P|_{\mathrm{vars}(t) \cup \mathrm{vars}(t')}$, $\sigma' = \sigma_Q|_{\mathrm{vars}(t) \cup \mathrm{vars}(t')}$, and $\Gamma'' = \Gamma_{\mathcal{N},\mathcal{K}} \cup (\Gamma|_{\mathrm{vars}(t) \cup \mathrm{vars}(t')})$. Since by assumption $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$, we have $\Gamma''_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma''_{\mathcal{X}} \to c''$ for some $c'' \subset c_\sigma$. Hence, by Lemma A.20, using Π, we have:

$$t\sigma\downarrow \neq \bot \iff t'\sigma'\downarrow \neq \bot$$

  i.e.

$$t\sigma_P^1\downarrow \neq \bot \iff t'\sigma_Q^1\downarrow \neq \bot$$

  Therefore, if rule Let-In is applied to P_i then it can also be applied to reduce Q_i into Q_i', and if the rule applied to P_i is Let-Else then it can also be applied to reduce Q_i into Q_i''. This proves point a). We prove here the Let-In case. The Let-Else case is similar (although slightly easier, since no new variable is added to the substitutions).

  In this case we have $\sigma_P^2 = \sigma_P^1 \cup \{t\sigma_P^1\downarrow /x\}$ and $\sigma_Q^2 = \sigma_Q^1 \cup \{t'\sigma_Q^1\downarrow /x\}$.

  In addition, by hypothesis, $t\sigma_P = t\sigma_P^1 = t\sigma$ and $t'\sigma_Q = t'\sigma_Q^1 = t'\sigma'$.

  By Lemma A.20, we know in this case that there exists $c \subseteq c''$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash t\sigma_P\downarrow \sim t'\sigma_Q\downarrow : T \to c$. Thus, by Lemma A.2, there exists $T' \in \mathrm{branches}(T)$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash t\sigma_P\downarrow \sim t'\sigma_Q\downarrow : T' \to c$.

  We choose $\Gamma' = \Gamma, x : T'$, $\sigma_P' = \sigma_P \cup \{t\sigma_P\downarrow /x\}$ and $\sigma_Q' = \sigma_Q \cup \{t'\sigma_Q\downarrow /x\}$. Since Γ does not contain union types, $\Gamma' \in \mathrm{branches}(\Gamma, x : T)$.

  Since $c \subseteq c'' \subseteq c_\sigma$ and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash t\sigma_P\downarrow \sim t'\sigma_Q\downarrow : T' \to c$, point b) holds.

  We now prove that point c) holds. Using Π', we have $\Gamma, x : T \vdash P_i' \sim Q_i' \to C_i'$. Hence, by Lemma A.7, there exists $C_i''' \subseteq C_i' (\subseteq C_i)$ such that $\Gamma' \vdash P_i' \sim Q_i' \to C_i'''$.

  Since $C_i''' \subseteq C_i$, we have

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i''' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} \subseteq [\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}.$$

  This last constraint set is consistent by hypothesis.

  Hence, by Lemma A.11, $[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i''' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is also consistent. This proves point c) and concludes this case.

- PLETDEC: then there exist y, P_i', P_i'', Q_i', Q_i'' such that $P_i = \texttt{let}\ x = \texttt{dec}(y, k_1)\ \texttt{in}\ P_i'\ \texttt{else}\ P_i''$, and $Q_i = \texttt{let}\ x = \texttt{dec}(y, k_2)\ \texttt{in}\ Q_i'\ \texttt{else}\ Q_i''$, and $\Gamma(y) = \texttt{LL}$. P_i reduces to either P_i' via

the Let-In rule, or P_i'' via the Let-Else rule. In addition

$$\Pi_i = \dfrac{\begin{array}{c}\Gamma(y) = \mathtt{LL} \\ \Gamma(k_1,k_2) = \mathrm{key}^{\mathtt{HH}}(T) \qquad \dfrac{\Pi_i'}{\Gamma, x:T \vdash P_i' \sim Q_i' \to C_i'} \qquad \dfrac{\Pi_i''}{\Gamma \vdash P_i'' \sim Q_i'' \to C_i''} \\ (\forall T'.\forall k_3 \neq k_2.\ \Gamma(k_1,k_3) = \mathrm{key}^{\mathtt{HH}}(T') \Rightarrow \dfrac{\Pi_i^{1,k_3}}{\Gamma, x:T' \vdash P_i' \sim Q_i'' \to C_{k_3}}) \\ (\forall T'.\forall k_3 \neq k_2.\ \Gamma(k_3,k_2) = \mathrm{key}^{\mathtt{HH}}(T') \Rightarrow \dfrac{\Pi_i^{2,k_3}}{\Gamma, x:T' \vdash P_i'' \sim Q_i' \to C'_{k_3}}) \end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{let}\ x = \mathtt{dec}(y,k_1)\ \mathtt{in}\ P_i'\ \mathtt{else}\ P_i'' \sim \mathtt{let}\ x = \mathtt{dec}(y,k_2)\ \mathtt{in}\ Q_i'\ \mathtt{else}\ Q_i'' \to \\ C_i' \cup C_i'' \cup (\bigcup_{k_3} C_{k_3}) \cup (\bigcup_{k_3} C'_{k_3})\end{array}}$$

We have $\alpha = \tau$. In addition, by hypothesis, $\sigma_P(y) = \sigma_P^1(y)$ and $\sigma_Q(y) = \sigma_Q^1(y)$.

We consider two cases.

– If $\mathtt{dec}(y\sigma_P, k_1)\downarrow \neq \bot$ then the reduction applied to P_i is Let-In, and P_i is reduced to P_i'. This also implies that there exists M such that $y\sigma_P = \mathtt{enc}(M, k_1)$. Since $\Gamma(y) = \mathtt{LL}$, we know by assumption that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash y\sigma_P \sim y\sigma_Q : \mathtt{LL} \to c$ for some constraint $c \subseteq c_\sigma$. Hence, by Lemma A.15, there exist N, k, T', $c' \subseteq c_\sigma$ such that $y\sigma_Q = \mathtt{enc}(N,k)$, $\Gamma(k_1,k) = \mathrm{key}^{\mathtt{HH}}(T)$, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M \sim N : T' \to c'$. Thus, by Lemma A.2, there exists $T'' \in \mathrm{branches}(T')$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M \sim N : T'' \to c'$.
Two cases are possible.

  * If $k = k_2$, then $\mathtt{dec}(y\sigma_Q, k_2)\downarrow = N$, and rule Let-In can be applied to reduce Q_i into Q_i', which proves point a). In this case we have $\sigma_P^2 = \sigma_P^1 \cup \{M/x\}$ and $\sigma_Q^2 = \sigma_Q^1 \cup \{N/x\}$.
  We choose $\Gamma' = \Gamma, x:T'$, $\sigma'_P = \sigma_P \cup \{M/x\}$ and $\sigma'_Q = \sigma_Q \cup \{N/x\}$ Since Γ does not contain union types, $\Gamma' \in \mathrm{branches}(\Gamma, x:T)$.
  Since $c' \subseteq c_\sigma$ and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M \sim N : T'' \to c'$, point b) holds.
  We now prove that point c) holds. Using Π_i', we have $\Gamma, x:T \vdash P_i' \sim Q_i' \to C_i'$. Hence, by Lemma A.7, there exists $C_i''' \subseteq C_i' (\subseteq C_i)$ such that $\Gamma' \vdash P_i' \sim Q_i' \to C_i'''$. Since $C_i''' \subseteq C_i$, we have

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i''' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} \subseteq [\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}.$$

  This last constraint set is consistent by hypothesis.
  Hence, by Lemma A.11, $[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i''' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is also consistent. This proves point c) and concludes this case.
  * If $k \neq k_2$, then $\mathtt{dec}(y\sigma_Q, k_2)\downarrow = \bot$, and rule Let-Else can be applied to reduce Q_i into Q_i'', which proves point a). In this case we have $\sigma_P^2 = \sigma_P^1 \cup \{M/x\}$ and $\sigma_Q'^2 = \sigma_Q^1$.
  We choose $\Gamma' = \Gamma, x:T'$, $\sigma'_P = \sigma_P \cup \{M/x\}$ and $\sigma'_Q = \sigma_Q \cup \{N/x\}$ (by well-formedness of the processes, x does not appear in Q_i''). Since Γ does not contain union types, $\Gamma' \in \mathrm{branches}(\Gamma, x:T)$.
  Since $c' \subseteq c_\sigma$ and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M \sim N : T'' \to c'$, point b) holds.
  We now prove that point c) holds. Using $\Pi_i^{1,k}$, we have $\Gamma, x:T \vdash P_i' \sim Q_i'' \to C_k$. Hence, by Lemma A.7, there exists $C_i''' \subseteq C_k (\subseteq C_i)$ such that $\Gamma' \vdash P_i' \sim Q_i'' \to C_i'''$.

Since $C_i''' \subseteq C_i$, we have

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i''' \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q} \subseteq [\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q}.$$

This last constraint set is consistent by hypothesis.
Hence, by Lemma A.11, $[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i''' \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q}$ is also consistent. This proves point c) and concludes this case.

– If $\mathtt{dec}(y\sigma_P, k_1)\downarrow = \bot$ then the reduction applied to P_i is Let-Else, and P_i is reduced to P_i''. Again we distinguish two cases.

* If $\mathtt{dec}(y\sigma_Q, k_2)\downarrow \neq \bot$ then rule Let-In can be applied to reduce Q_i into Q_i'. This also implies that there exists N such that $y\sigma_Q = \mathtt{enc}(N, k_2)$. Since $\Gamma(y) = \mathtt{LL}$, we know by assumption that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash y\sigma_P \sim y\sigma_Q : \mathtt{LL} \to c$ for some constraint $c \subseteq c_\sigma$. Hence, by Lemma A.15, there exist M, k, T', $c' \subseteq c_\sigma$ such that $y\sigma_P = \mathtt{enc}(M, k)$, $\Gamma(k, k_2) = \mathrm{key}^{\mathtt{HH}}(T)$, and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M \sim N : T' \to c'$. Thus, by Lemma A.2, there exists $T'' \in \mathrm{branches}(T')$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M \sim N : T'' \to c'$.
In this case we have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q'^2 = \sigma_Q^1 \cup \{N/x\}$. We choose $\Gamma' = \Gamma, x : T'$, $\sigma_P' = \sigma_P \cup \{M/x\}$ and $\sigma_Q' = \sigma_Q \cup \{N/x\}$ (by well-formedness of the processes, x does not appear in P_i''). Since Γ does not contain union types, $\Gamma' \in \mathrm{branches}(\Gamma, x : T)$.
Since $c' \subseteq c_\sigma$ and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M \sim N : T'' \to c'$, point b) holds.
We now prove that point c) holds. Using $\Pi_i^{2,k}$, we have $\Gamma, x : T \vdash P_i'' \sim Q_i' \to C_k'$. Hence, by Lemma A.7, there exists $C_i''' \subseteq C_k' (\subseteq C_i)$ such that $\Gamma' \vdash P_i'' \sim Q_i' \to C_i'''$. Since $C_i''' \subseteq C_i$, we have

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i''' \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q} \subseteq [\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q}.$$

This last constraint set is consistent by hypothesis.
Hence, by Lemma A.11, $[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i''' \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q}$ is also consistent. This proves point c) and concludes this case.

* If $\mathtt{dec}(y\sigma_Q, k_2)\downarrow = \bot$ then rule Let-Else can be applied to reduce Q_i into Q_i''. In this case we have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q'^2 = \sigma_Q^1$. We choose $\Gamma' = \Gamma$, $\sigma_P' = \sigma_P$ and $\sigma_Q' = \sigma_Q$. Since the substitutions and environments do not change, point b) clearly holds.
We now prove that point c) holds. Using Π_i'', we have $\Gamma \vdash P_i'' \sim Q_i'' \to C_i''$. Since $C_i'' \subseteq C_i$, we have

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i'' \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q} \subseteq [\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q}.$$

This last constraint set is consistent by hypothesis.
Hence, by Lemma A.11, $[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i'' \cup_\forall c_\phi]\!]_{\sigma_P, \sigma_Q}$ is also consistent. This proves point c) and concludes this case.

- PLetAdecSame and PLetAdecDiff: these two cases are similar to the PLetDec case.
- PLetLR: then $P_i = \mathtt{let}\ x = d(y)\ \mathtt{in}\ P_i'\ \mathtt{else}\ P_i''$ and $Q_i = \mathtt{let}\ x = d(y)\ \mathtt{in}\ Q_i'\ \mathtt{else}\ Q_i''$ for some P_i', P_i'', Q_i', Q_i''.

  P_i reduces to either P_i' via the Let-In rule, or P_i'' via the Let-Else rule.

In addition

$$\Pi_i = \frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma(y) = [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!] \vee \Gamma(y) <: \mathrm{key}^{l'}(T) \qquad \dfrac{\Pi''}{\Gamma \vdash P_i'' \sim Q_i'' \to C_i''}}{\Gamma \vdash P_i \sim Q_i \to C_i (= C_i'')}.$$

We have $\alpha = \tau$. By assumption we also have $\sigma_P(y) = \sigma_P^1(y)$ and $\sigma_Q(y) = \sigma_Q^1(y)$.

By hypothesis, σ_P, σ_Q are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$. Hence, by definition of the well-typedness of substitutions, there exists $c_y \subseteq c_\sigma$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P(y) \sim \sigma_Q(y) : [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!] \to c_y$, or $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P(y) \sim \sigma_Q(y) : \mathrm{key}^{l'}(T) \to c_y$ In the first case by Lemma A.14, $\sigma_P(y) = m$ and $\sigma_Q(y) = n$. Since m, n are nonces, $d(m)\!\downarrow = d(n)\!\downarrow = \bot$, and we thus have $d(\sigma_P(y))\!\downarrow = d(\sigma_Q(y))\!\downarrow = \bot$. Similarly, in the second case, by Lemma A.18, $\sigma_P(y)$ and $\sigma_Q(y)$ are both keys in $\mathcal{K}$, and thus $d(\sigma_P(y))\!\downarrow = d(\sigma_Q(y))\!\downarrow = \bot$.

Therefore the reduction rule applied to P_i can only be Let-Else, and P_i is reduced to P_i''. Since we also have $d(\sigma_Q(y))\!\downarrow = \bot$, this rule can also be applied to reduce Q_i into Q_i''. This proves point a).

We therefore have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q^2 = \sigma_Q^1$. We choose $\Gamma' = \Gamma$.

Since the substitutions and typing environments are unchanged by the reduction, point b) clearly holds.

Moreover, Π'', and the fact that

$$[\![(\cup_{\times\, j \neq i} C_j) \cup_\times C_i'' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} = [\![(\cup_{\times\, j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$$

which is consistent by hypothesis, prove point c) and conclude this case.

- PIfL: then $P_i = \texttt{if}\ M = M'\ \texttt{then}\ P_i^\top\ \texttt{else}\ P_i^\bot$ and $Q_i = \texttt{if}\ N = N'\ \texttt{then}\ Q_i^\top\ \texttt{else}\ Q_i^\bot$ for some $Q_i^\top$, $Q_i^\bot$. P_i reduces to P_i' which is either $P_i^\top$ via the If-Then rule, or $P_i^\bot$ via the If-Else rule. In addition

$$\Pi_i = \frac{\dfrac{\Pi^\top}{\Gamma \vdash P_i^\top \sim Q_i^\top \to C_i^\top} \qquad \dfrac{\Pi^\bot}{\Gamma \vdash P_i^\bot \sim Q_i^\bot \to C_i^\bot} \qquad \dfrac{\Pi}{\Gamma \vdash M \sim N : \mathtt{LL} \to c} \qquad \dfrac{\Pi'}{\Gamma \vdash M' \sim N' : \mathtt{LL} \to c'}}{\Gamma \vdash P_i \sim Q_i \to C_i = (C_i^\top \cup C_i^\bot) \cup_\forall (c \cup c')}$$

We have $\alpha = \tau$. In addition, by hypothesis, $t\sigma_P = t\sigma_P^1$ for $t \in \{M, M'\}$ and $t'\sigma_Q = t'\sigma_Q^1$ for $t' \in \{N, N'\}$.

Since $\Gamma \vdash M \sim N : \mathtt{LL} \to c$, by Lemma A.22, there exists $c'' \subseteq [\![c]\!]_{\sigma_P,\sigma_Q} \cup c_\sigma$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma_P \sim N\sigma_Q : \mathtt{LL} \to c''$. Similarly, there exists $c''' \subseteq [\![c']\!]_{\sigma_P,\sigma_Q} \cup c_\sigma$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M'\sigma_P \sim N'\sigma_Q : \mathtt{LL} \to c'''$.

Hence, by Lemma A.19, either $M\sigma_P\!\downarrow = N\sigma_Q\!\downarrow = \bot$; or $M\sigma_P\!\downarrow = M\sigma_P \neq \bot$ and $N\sigma_Q\!\downarrow = N\sigma_Q \neq \bot$. Similarly, either $M'\sigma_P\!\downarrow = N'\sigma_Q\!\downarrow = \bot$; or $M'\sigma_P\!\downarrow = M'\sigma_P \neq \bot$ and $N'\sigma_Q\!\downarrow = N'\sigma_Q \neq \bot$.

Let us first consider the case where $M\sigma_P\!\downarrow \neq \bot$, $M'\sigma_P\!\downarrow \neq \bot$, $N\sigma_Q\!\downarrow \neq \bot$ and $N'\sigma_Q\!\downarrow \neq \bot$.

Let $\phi = \{M\sigma_P/x, M'\sigma_P/y\}$ and $\phi' = \{N\sigma_Q/x, N'\sigma_Q/y\}$. We then have $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \phi \sim \phi' : \mathtt{LL} \to c'' \cup c'''$.

Let us prove that $c \cup c'''$ is consistent in some typing environment. By hypothesis, σ_P, σ_Q are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$. Hence, by Lemma A.4, there exists $\Gamma'' \in \text{branches}(\Gamma)$ such that $\Gamma''_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma''_{\mathcal{X}} \to c_\sigma$. By Lemma A.13, there exists $(c_1, \Gamma''') \in C_i$ such that $\Gamma'' \subseteq \Gamma'''$. Since $C_i = (C_i^\top \cup C_i^\perp) \cup_\forall (c \cup c')$, c_1 is of the form $c_2 \cup c \cup c'$ for some c_2.

As we noted previously, $[\![C_i]\!]_{\sigma_P,\sigma_Q} \cup_\forall c_\sigma$ is consistent. Therefore, by Lemma A.11, the constraint $\{([\![c \cup c']\!]_{\sigma_P,\sigma_Q} \cup c_\sigma, \Gamma''')\}$ is consistent. Hence, by the same Lemma, $c'' \cup c'''$ is also consistent in Γ'''.

Thus, by Lemma A.25, ϕ and ϕ' are statically equivalent. Hence, in particular, $M\sigma_P = M'\sigma_P \iff N\sigma_Q = N'\sigma_Q$.

Therefore, if rule If-Then is applied to P_i then it can also be applied to reduce Q_i into $Q_i^\top$, and if the rule applied to P_i is If-Else then it can also be applied to reduce Q_i into $Q_i^\perp$. This proves point a). We prove here the If-Then case. The If-Else case is similar.

We choose $\Gamma' = \Gamma$. We have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q^2 = \sigma_Q^1$.

Since the substitutions and environments do not change in this reduction, point b) trivially holds.

Moreover, by hypothesis,

$$\left[\!\!\left[(\cup_{\times j \neq i} C_j) \cup_\times (C_i^\top \cup C_i^\perp) \cup_\forall (c \cup c' \cup c_\phi)\right]\!\!\right]_{\sigma_P,\sigma_Q}$$

is consistent. Thus by Lemma A.11,

$$\left[\!\!\left[(\cup_{\times j \neq i} C_j) \cup_\times (C_i^\top \cup C_i^\perp) \cup_\forall c_\phi\right]\!\!\right]_{\sigma_P,\sigma_Q}$$

is also consistent. Since, using $C_i' = C_i^\top$ and $C_i = (C_i^\top \cup C_i^\perp) \cup_\forall (c \cup c')$,

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} \subseteq \left[\!\!\left[(\cup_{\times j \neq i} C_j) \cup_\times (C_i^\top \cup C_i^\perp) \cup_\forall c_\phi\right]\!\!\right]_{\sigma_P,\sigma_Q},$$

we have by Lemma A.11 that $[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is consistent. $\Pi^\top$ and this fact prove point c) and conclude this case.

The case where $M\sigma_P{\downarrow} = N\sigma_Q{\downarrow} = \perp$ or $M'\sigma_P{\downarrow} = N'\sigma_Q{\downarrow} = \perp$ remains. In that case, the rule applied to P_i is necessarily $If - Else$, and this rule can also be applied to Q_i. We conclude the proof similarly to the previous case.

- PIFLR: $P_i = \mathtt{if}\ M_1 = M_2\ \mathtt{then}\ P_i^\top\ \mathtt{else}\ P_i^\perp$ and $Q_i = \mathtt{if}\ N_1 = N_2\ \mathtt{then}\ Q_i^\top\ \mathtt{else}\ Q_i^\perp$ for some $Q_i^\top$, $Q_i^\perp$. P_i reduces to P_i' which is either $P_i^\top$ via the If-Then rule, or $P_i^\perp$ via the If-Else rule. In addition

$$\Pi_i = \dfrac{b = (\tau_m^{l,1} \stackrel{?}{=} \tau_{m'}^{l'',1}) \qquad b' = (\tau_n^{l',1} \stackrel{?}{=} \tau_{n'}^{l''',1}) \qquad \dfrac{\Pi''}{\Gamma \vdash P_i^b \sim Q_i^{b'} \to C_i'} \qquad \dfrac{\Pi}{\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,1}\,;\, \tau_n^{l',1}]\!] \to \emptyset} \qquad \dfrac{\Pi'}{\Gamma \vdash M_2 \sim N_2 : [\![\tau_{m'}^{l'',1}\,;\, \tau_{n'}^{l''',1}]\!] \to c'}}{\Gamma \vdash P_i \sim Q_i \to C_i = C_i'}$$

We have $\alpha = \tau$ in any case. In addition, by assumption, $t\sigma_P = t\sigma_P^1$ for $t \in \{M_1, M_2\}$ and $t'\sigma_Q = t'\sigma_Q^1$ for $t' \in \{N_1, N_2\}$.

By hypothesis, σ_P, σ_Q are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$. Hence, by Lemma A.22, using Π, there exists c'' such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M_1\sigma_P \sim N_1\sigma_Q : [\![\tau_m^{l,1} ; \tau_n^{l',1}]\!] \to c''$. Therefore by Lemma A.14, $M_1\sigma_P = m$ and $N_1\sigma_Q = n$. Similarly we can show that $M_2\sigma_P = m'$ and $N_2\sigma_Q = n'$.

There are four cases for b and b', which are all similar. We write the proof for the case where $b = \top$ and $b' = \bot$, *i.e.* $\tau_m^{l,1} = \tau_{m'}^{l'',1}$ and $\tau_n^{l',1} \neq \tau_{n'}^{l''',1}$.

Thus the reduction rule applied to P_i is If-Then and $P_i' = P_i^\top$. On the other hand, rule If-Else can be applied to reduce Q_i into $Q_i' = Q_i^\bot$. This proves point a) (these rules both correspond to silent actions).

We choose $\Gamma' = \Gamma$. We have $\sigma_P' = \sigma_P$ and $\sigma_Q' = \sigma_Q$.

Since the substitutions and environments do not change in this reduction, point b) trivially holds.

Moreover, Π'' and the fact that

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} = [\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$$

prove point c) and conclude this case.

- PIFS: then $P_i = \texttt{if } M = M' \texttt{ then } P_i^\top \texttt{ else } P_i^\bot$ and $Q_i = \texttt{if } N = N' \texttt{ then } Q_i^\top \texttt{ else } Q_i^\bot$ for some $Q_i^\top$, $Q_i^\bot$. P_i reduces to P_i' which is either $P_i^\top$ via the If-Then rule, or $P_i^\bot$ via the If-Else rule. In addition

$$\Pi_i = \frac{\Pi^\bot \qquad \Pi \qquad \Pi'}{\dfrac{\Gamma \vdash P_i^\bot \sim Q_i^\bot \to C_i' \qquad \Gamma \vdash M \sim N : \texttt{LL} \to c \qquad \Gamma \vdash M' \sim N' : \texttt{HH} \to c'}{\Gamma \vdash P_i \sim Q_i \to C_i = C_i'}}$$

We have $\alpha = \tau$ in any case. In addition, by hypothesis, $t\sigma_P = t\sigma_P^1$ for $t \in \{M, M'\}$ and $t'\sigma_Q = t'\sigma_Q^1$ for $t' \in \{N, N'\}$.

By hypothesis, σ_P, σ_Q are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$. Hence, by Lemma A.22, using Π, there exists c'' such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma_P \sim N\sigma_Q : \texttt{LL} \to c''$. Similarly we can show that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M'\sigma_P \sim N'\sigma_Q : \texttt{HH} \to c'''$ for some c'''.

Hence, by Lemma A.19, either $M\sigma_P{\downarrow_=}\, N\sigma_Q{\downarrow_=}\, \bot$; or $M\sigma_P{\downarrow_=}\, M\sigma_P \neq \bot$ and $N\sigma_Q{\downarrow_=}\, N\sigma_Q \neq \bot$. Similarly, either $M'\sigma_P{\downarrow_=}\, N'\sigma_Q{\downarrow_=}\, \bot$; or $M'\sigma_P{\downarrow_=}\, M'\sigma_P \neq \bot$ and $N'\sigma_Q{\downarrow_=}\, N'\sigma_Q \neq \bot$.

Let us first consider the case where $M\sigma_P{\downarrow}\neq \bot$, $M'\sigma_P{\downarrow}\neq \bot$, $N\sigma_Q{\downarrow}\neq \bot$ and $N'\sigma_Q{\downarrow}\neq \bot$.

Therefore by Lemma A.23, $M\sigma_P \neq M'\sigma_P$ and $N\sigma_Q \neq N'\sigma_Q$. Hence the reduction for P_i is necessarily If-Else, which is also applicable to reduce Q_i to $Q_i^\bot$. This proves point a).

We choose $\Gamma' = \Gamma$. We have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q^2 = \sigma_Q^1$.

Since the substitutions and typing environments do not change in this reduction, point b) trivially holds.

Moreover, Π'' and the fact that

$$[\![(\cup_{\times j \neq i} C_j) \cup_\times C_i' \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q} = [\![(\cup_{\times j} C_j) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$$

prove point c) and conclude this case.

The case where $M\sigma_P{\downarrow} = N\sigma_Q{\downarrow} = \bot$ or $M'\sigma_P{\downarrow} = N'\sigma_Q{\downarrow} = \bot$ remains. In that case, the rule applied to P_i is necessarily $If - Else$, and this rule can also be applied to Q_i. We conclude the proof similarly to the previous case.

- PIfI: then $P_i = \texttt{if } M = M' \texttt{ then } P_i^\top \texttt{ else } P_i^\bot$ and $Q_i = \texttt{if } N = N' \texttt{ then } Q_i^\top \texttt{ else } Q_i^\bot$ for some $Q_i^\top$, $Q_i^\bot$. This case is similar to the PIfS case: the incompatibility of the types of M, N and M', N' ensures that the processes can only follow the else branch.

  P_i reduces to P_i' which is either $P_i^\top$ via the If-Then rule, or $P_i^\bot$ via the If-Else rule. In addition

$$\Pi_i = \frac{\dfrac{\Pi^\bot}{\Gamma \vdash P_i^\bot \sim Q_i^\bot \to C_i'} \qquad \dfrac{\Pi}{\Gamma \vdash M \sim N : T * T' \to c} \qquad \dfrac{\Pi'}{\Gamma \vdash M' \sim N' : [\![\tau_m^{l,a}\,;\, \tau_n^{l',a}]\!] \to c'}}{\Gamma \vdash P_i \sim Q_i \to C_i = C_i'}$$

  We have $\alpha = \tau$ in any case. In addition, by hypothesis, $t\sigma_P = t\sigma_P^1$ for $t \in \{M, M'\}$ and $t'\sigma_Q = t'\sigma_Q^1$ for $t' \in \{N, N'\}$.

  By hypothesis, σ_P, σ_Q are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$. Hence, by Lemma A.22, using Π, there exists c'' such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma_P \sim N\sigma_Q : T * T' \to c''$.

  Hence, by Lemma A.19, either $M\sigma_P{\downarrow} = N\sigma_Q{\downarrow} = \bot$; or $M\sigma_P{\downarrow} = M\sigma_P \neq \bot$ and $N\sigma_Q{\downarrow} = N\sigma_Q \neq \bot$. Let us first consider the case where $M\sigma_P{\downarrow} \neq \bot$, and $N\sigma_Q{\downarrow} \neq \bot$.

  By Lemma A.17, $M\sigma_P$ and $N\sigma_Q$ both are pairs. Similarly we can show that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M'\sigma_P \sim N'\sigma_Q : [\![\tau_m^{l,a}\,;\, \tau_n^{l',a}]\!] \to c'''$ for some c'''. By Lemma A.14, this implies that $M'\sigma_P = m$ and $N'\sigma_Q = n$. Thus neither of these two terms are pairs.

  Therefore $M\sigma_P \neq M'\sigma_P$ and $N\sigma_Q \neq N'\sigma_Q$. The end of the proof for this case is then the same as for the PIfS case.

  The case where $M\sigma_P{\downarrow} = N\sigma_Q{\downarrow} = \bot$ or $M'\sigma_P{\downarrow} = N'\sigma_Q{\downarrow} = \bot$ remains. In that case, the rule applied to P_i is necessarily $If - Else$, and this rule can also be applied to Q_i. We conclude the proof similarly to the previous case.

- PIfP: then $P_i = \texttt{if } M = t \texttt{ then } P_i^\top \texttt{ else } P_i^\bot$ and $Q_i = \texttt{if } N = t \texttt{ then } Q_i^\top \texttt{ else } Q_i^\bot$ for some $Q_i^\top$, $Q_i^\bot$, some messages M, N, and some $t \in C \cup \mathcal{K} \cup \mathcal{N}$. P_i reduces to P_i' which is either $P_i^\top$ via the If-Then rule, or $P_i^\bot$ via the If-Else rule. In addition

$$\Pi_i = \frac{\dfrac{\dfrac{\Pi^\top}{\Gamma \vdash P_i^\top \sim Q_i^\top \to C_i^\top} \qquad \dfrac{\Pi^\bot}{\Gamma \vdash P_i^\bot \sim Q_i^\bot \to C_i^\bot}}{\Gamma \vdash M \sim N : \mathtt{LL} \to c} \qquad \dfrac{\Pi'}{\Gamma \vdash t \sim t : \mathtt{LL} \to c'} \qquad t \in \mathcal{C} \cup \mathcal{K} \cup \mathcal{N}}{\Gamma \vdash P_i \sim Q_i \to C_i = C_i^\top \cup C_i^\bot}$$

  We have in any case $\alpha = \tau$. In addition, by assumption, $t'\sigma_P = t'\sigma_P^1$ for $t' \in \{M, M'\}$ and $t'\sigma_Q = t'\sigma_Q^1$ for $t' \in \{N, N'\}$.

  By hypothesis, σ_P, σ_Q are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_{\mathcal{X}} \to c_\sigma$. Hence, by Lemma A.22, using Π, there exists $c'' \subseteq [\![c]\!]_{\sigma_P,\sigma_Q} \cup c_\sigma$ such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M\sigma_P \sim N\sigma_Q : \mathtt{LL} \to c''$.

Hence, by Lemma A.19, either $M\sigma_P{\downarrow} = N\sigma_Q{\downarrow} = \bot$; or $M\sigma_P{\downarrow} = M\sigma_P \neq \bot$ and $N\sigma_Q{\downarrow} = N\sigma_Q \neq \bot$. Similarly, either $t{\downarrow} = \bot$ or $t{\downarrow} = t \neq \bot$.

Let us first consider the case where $M\sigma_P{\downarrow} \neq \bot$, $M'\sigma_P{\downarrow} \neq \bot$, and $t{\downarrow} \neq \bot$.

We then show that $M\sigma_P = t$ if and only if $N\sigma_Q = t$ (note that since t is ground, $t = t\sigma_P = t\sigma_Q$). If $M\sigma_P = t$, then $\Gamma_{\mathcal{N},\mathcal{K}} \vdash t \sim N\sigma_Q : \mathtt{LL} \to c''$. In all possible cases for t, *i.e.* $t \in \mathcal{K}$, $t \in \mathcal{N}$, and $t \in \mathcal{C}$, Lemma A.18 implies that $N\sigma_Q = t$. This proves the first direction of the equivalence, the other direction is similar.

Therefore, if rule If-Then is applied to P_i then it can also be applied to reduce Q_i into $Q_i^\top$, and if the rule applied to P_i is If-Else then it can also be applied to reduce Q_i into $Q_i^\bot$. This proves point a). We prove here the If-Then case. The If-Else case is similar.

We choose $\Gamma' = \Gamma$. We have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q^2 = \sigma_Q^1$.

Since the substitutions and typing environments do not change in this reduction, point b) trivially holds.

Moreover, by hypothesis,

$$\left[\!\left[(\cup_{\times j\neq i} C_j)\cup_\times (C_i^\top \cup C_i^\bot)\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q}$$

is consistent. Since, using $C_i' = C_i^\top$ and $C_i = (C_i^\top \cup C_i^\bot)$, we have

$$\left[\!\left[(\cup_{\times j\neq i} C_j)\cup_\times C_i'\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q} \subseteq \left[\!\left[(\cup_{\times j\neq i} C_j)\cup_\times (C_i^\top \cup C_i^\bot)\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q},$$

we have by Lemma A.11 that $\left[\!\left[(\cup_{\times j\neq i} C_j)\cup_\times C_i'\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q}$ is consistent. This fact proves point c) and concludes this case.

The case where $M\sigma_P{\downarrow} = N\sigma_Q{\downarrow} = \bot$ or $t{\downarrow} = \bot$ remains. In that case, the rule applied to P_i is necessarily $If - Else$, and this rule can also be applied to Q_i. We conclude the proof similarly to the previous case.

- PIFLR*: then $P_i = \mathtt{if}\ M_1 = M_2\ \mathtt{then}\ P_i^\top\ \mathtt{else}\ P_i^\bot$ and $Q_i = \mathtt{if}\ N_1 = N_2\ \mathtt{then}\ Q_i^\top\ \mathtt{else}\ Q_i^\bot$ for some $Q_i^\top$, $Q_i^\bot$. P_i reduces to P_i' which is either $P_i^\top$ via the If-Then rule, or $P_i^\bot$ via the If-Else rule. In addition

$$\Pi_i = \dfrac{\dfrac{\Pi'}{\Gamma \vdash M_2 \sim N_2 : [\![\tau_m^{l,\infty}\,;\tau_n^{l',\infty}]\!] \to c_2} \quad \dfrac{\dfrac{\Pi}{\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,\infty}\,;\tau_n^{l',\infty}]\!] \to c_1}}{\Gamma \vdash P_i^\top \sim Q_i^\top \to C_i^\top} \quad \dfrac{\Pi^\bot}{\Gamma \vdash P_i^\bot \sim Q_i^\bot \to C_i^\bot}}{\Gamma \vdash P_i \sim Q_i \to C_i = C_i^\top \cup C_i^\bot}.$$

We have $\alpha = \tau$ in any case. In addition, by assumption, $t\sigma_P = t\sigma_P^1$ for $t \in \{M_1, M_2\}$ and $t'\sigma_Q = t'\sigma_Q^1$ for $t' \in \{N_1, N_2\}$.

By hypothesis, σ_P, σ_Q are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_\mathcal{X} \to c_\sigma$. Hence, by Lemma A.22, using Π, there exists c'' such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M_1\sigma_P \sim N_1\sigma_Q : [\![\tau_m^{l,\infty}\,;\tau_m^{l',\infty}]\!] \to c''$. Therefore by Lemma A.14, $M_1\sigma_P = m$ and $N_1\sigma_Q = n$. Similarly we can show that $M_2\sigma_P = m$ and $N_2\sigma_Q = n$.

Hence $M_1' = M_2'$ and $N_1' = N_2'$.

Thus the reduction rule applied to P_i is If-Then and $P_i' = P_i^\top$. On the other hand, rule If-Then can also be applied to reduce Q_i into $Q_i' = Q_i^\top$. This proves point a).

Note that we still need to type the other branch, even though it is not used here, as when replicating the process this test may fail if M_1, N_1 and M_2, N_2 are nonces from different sessions.

We choose $\Gamma' = \Gamma$. We have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q^2 = \sigma_Q^1$.

Since the substitutions and environments do not change in this reduction, point b) trivially holds.

Moreover, Π'' and the fact that, with $C_i' = C_i^\top$,

$$\begin{aligned}\left[\!\left[(\cup_{\times j\neq i} C_j)\cup_\times C_i'\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q} &\subseteq \left[\!\left[(\cup_{\times j\neq i} C_j)\cup_\times (C_i^\top \cup C_i^\bot)\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q}\\ &= \left[\!\left[(\cup_{\times j} C_j)\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q}\end{aligned}$$

prove point c) and conclude this case.

- PIfLR'*: $P_i = \texttt{if } M_1 = M_2 \texttt{ then } P_i^\top \texttt{ else } P_i^\bot$ and $Q_i = \texttt{if } N_1 = N_2 \texttt{ then } Q_i^\top \texttt{ else } Q_i^\bot$ for some $Q_i^\top$, $Q_i^\bot$. P_i reduces to P_i' which is either $P_i^\top$ via the If-Then rule, or $P_i^\bot$ via the If-Else rule. In addition

$$\Pi_i = \dfrac{\dfrac{\Pi}{\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!] \to c_1} \quad \dfrac{\Pi'}{\Gamma \vdash M_2 \sim N_2 : [\![\tau_{m'}^{l'',a}\,;\,\tau_{n'}^{l''',a}]\!] \to c_2} \quad \tau_m^{l,a} \neq \tau_{m'}^{l'',a} \quad \tau_n^{l',a} \neq \tau_{n'}^{l''',a} \quad \dfrac{\Pi''}{\Gamma \vdash P_i^\bot \sim Q_i^\bot \to C_i'}}{\Gamma \vdash P_i \sim Q_i \to C_i = C_i'}$$

We have $\alpha = \tau$ in any case. In addition, by assumption, $t\sigma_P = t\sigma_P^1$ for $t \in \{M_1, M_2\}$ and $t'\sigma_Q = t'\sigma_Q^1$ for $t' \in \{N_1, N_2\}$.

By hypothesis, σ_P, σ_Q are ground and $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma_P \sim \sigma_Q : \Gamma_\mathcal{X} \to c_\sigma$. Hence, by Lemma A.22, using Π, there exists c'' such that $\Gamma_{\mathcal{N},\mathcal{K}} \vdash M_1\sigma_P \sim N_1\sigma_Q : [\![\tau_m^{l,a}\,;\,\tau_n^{l',a}]\!] \to c''$. Therefore by Lemma A.14, $M_1\sigma_P = m$ and $N_1\sigma_Q = n$. Similarly, using Lemma A.14, we can show that $M_2\sigma_P = m'$ and $N_2\sigma_Q = n'$.

Moreover, since $\tau_m^{l,a} \neq \tau_{m'}^{l'',a}$, we know that $m \neq m'$ (by well-formedness of the processes), and similarly $n \neq n'$.

Hence, $M_1\sigma_P \neq M_2\sigma_P$ and $N_1\sigma_Q \neq N_2\sigma_Q$.

Thus the reduction rule applied to P_i is If-Else and $P_i' = P_i^\bot$. On the other hand, rule If-Else can also be applied to reduce Q_i into $Q_i' = Q_i^\bot$. This proves point a).

We choose $\Gamma' = \Gamma$. We have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q^2 = \sigma_Q^1$.

Since the substitutions and environments do not change in this reduction, point b) trivially holds.

Moreover, Π'' and the fact that

$$\left[\!\left[(\cup_{\times j\neq i} C_j)\cup_\times C_i'\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q} = \left[\!\left[(\cup_{\times j} C_j)\cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q}$$

prove point c) and conclude this case.

- PIFALL: then $P_i = \texttt{if}\ M = M'\ \texttt{then}\ P_i^\top\ \texttt{else}\ P_i^\bot$ and $Q_i = \texttt{if}\ N = N'\ \texttt{then}\ Q_i^\top\ \texttt{else}\ Q_i^\bot$ for some $Q_i^\top$, $Q_i^\bot$. P_i reduces to P_i' which is either $P_i^\top$ via the If-Then rule, or $P_i^\bot$ via the If-Else rule. In addition

$$\Pi_i = \dfrac{\dfrac{\Pi^{\top,\bot}}{\Gamma \vdash P_i^\top \sim Q_i^\bot \to C_i^{\top,\bot}} \quad \dfrac{\dfrac{\Pi^{\top,\top}}{\Gamma \vdash P_i^\top \sim Q_i^\top \to C_i^{\top,\top}}}{\Pi^{\bot,\top}} \quad \dfrac{\Pi^{\bot,\bot}}{\Gamma \vdash P_i^\bot \sim Q_i^\bot \to C_i^{\bot,\bot}}}{\begin{array}{c}\Gamma \vdash \texttt{if}\ M = M'\ \texttt{then}\ P_i^\top\ \texttt{else}\ P_i^\bot \sim \texttt{if}\ N = N'\ \texttt{then}\ Q_i^\top\ \texttt{else}\ Q_i^\bot \to \\ C_i = C_i^{\top,\top} \cup C_i^{\top,\bot} \cup C_i^{\bot,\top} \cup C_i^{\bot,\bot}\end{array}}$$

  In addition, by hypothesis, $t\sigma_P = t\sigma_P^1$ for $t \in \{M, M'\}$ and $t'\sigma_Q = t'\sigma_Q^1$ for $t' \in \{N, N'\}$.

  Four cases are possible:

  - $M\sigma_P{\downarrow} \neq \bot$, $M'\sigma_P{\downarrow} \neq \bot$, $N\sigma_Q{\downarrow} \neq \bot$, $N'\sigma_Q{\downarrow} \neq \bot$, $M\sigma_P = M'\sigma_P$ and $N\sigma_Q = N'\sigma_Q$;
  - or $M\sigma_P{\downarrow} \neq \bot$, $M'\sigma_P{\downarrow} \neq \bot$, $M\sigma_P = M'\sigma_P$ and ($N\sigma_Q \neq N'\sigma_Q$ or $N\sigma_Q{\downarrow} = \bot$ or $N'\sigma_Q{\downarrow} = \bot$);
  - or $N\sigma_Q{\downarrow} \neq \bot$, $N'\sigma_Q{\downarrow} \neq \bot$, $N\sigma_Q = N'\sigma_Q$ and ($M\sigma_P \neq M'\sigma_P$ or $M\sigma_P{\downarrow} = \bot$ or $M'\sigma_P{\downarrow} = \bot$);
  - or ($M\sigma_P \neq M'\sigma_P$ or $M\sigma_P{\downarrow} = \bot$ or $M'\sigma_P{\downarrow} = \bot$) and ($N\sigma_Q \neq N'\sigma_Q$ or $N\sigma_Q{\downarrow} = \bot$ or $N'\sigma_Q{\downarrow} = \bot$).

  In any case, we have $\alpha = \tau$. These four cases are similar, we detail the proof for the second case, where $M\sigma_P = M'\sigma_P$ and $N\sigma_Q \neq N'\sigma_Q$.

  Since $M\sigma_P = M'\sigma_P$, $M'\sigma_P{\downarrow} \neq \bot$, and $M\sigma_P = M'\sigma_P$, the reduction applied to P_i can only be If-Then, and P_i is reduced to $P_i^\top$. Since $N\sigma_Q \neq N'\sigma_Q$, $N\sigma_Q{\downarrow} = \bot$, or $N'\sigma_Q{\downarrow} = \bot$, rule If-Else can be applied to reduce Q_i into $Q_i^\bot$. This proves point a).

  We choose $\Gamma' = \Gamma$. We have $\sigma_P^2 = \sigma_P^1$ and $\sigma_Q^2 = \sigma_Q^1$.

  Since the substitutions and environments do not change in this reduction, point b) trivially holds.

  Moreover, by hypothesis,

$$\left[\!\left[(\cup_{\times j \neq i} C_j) \cup_\times (C_i^{\top,\top} \cup C_i^{\top,\bot} \cup C_i^{\bot,\top} \cup C_i^{\bot,\bot}) \cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q}$$

  is consistent. Since, using $C_i' = C_i^{\top,\bot}$,

$$\left[\!\left[(\cup_{\times j \neq i} C_j) \cup_\times C_i' \cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q} \subseteq \left[\!\left[(\cup_{\times j \neq i} C_j) \cup_\times (C_i^{\top,\top} \cup C_i^{\top,\bot} \cup C_i^{\bot,\top} \cup C_i^{\bot,\bot}) \cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q},$$

  we have by Lemma A.11 that $\left[\!\left[(\cup_{\times j \neq i} C_j) \cup_\times C_i' \cup_\forall c_\phi\right]\!\right]_{\sigma_P,\sigma_Q}$ is consistent. $\Pi^{\top,\bot}$ and this fact prove point c) and conclude this case.

□

Theorem A.1 (Typing implies trace inclusion)**.** *For all processes P, Q, for all ϕ_P, ϕ_Q, σ_P, σ_Q, for all multisets of processes $\mathcal{P}$, $\mathcal{Q}$ for all constraints C, for all sequence s of actions, for all Γ containing only keys, if*

$$\Gamma \vdash P \sim Q \to C,$$

and if C is consistent, then

$$P \sqsubseteq_t Q$$

that is, if

$$(\{P\}, \emptyset, \emptyset) \xrightarrow{s}_* (\mathcal{P}, \phi_P, \sigma_P),$$

then there exists a sequence s' of actions, a multiset $\mathcal{Q}$, a frame ϕ_Q, a substitution σ_Q, such that

- $s =_\tau s'$
- $(\{Q\}, \emptyset, \emptyset) \xrightarrow{s'}_* (\mathcal{Q}, \phi_Q, \sigma_Q)$,
- *$\phi_P\sigma_P$ and $\phi_Q\sigma_Q$ are statically equivalent.*

Proof. We successively apply Lemma A.26 to each of the reduction steps in the reduction

$$(\{P\}, \emptyset, \emptyset) \xrightarrow{s}_* (\mathcal{P}, \phi_P, \sigma_P).$$

The lemma can indeed be applied successively. At each reduction step of P we obtain a sequence of reduction steps for Q with the same actions, and the conclusions the lemma provides imply the conditions needed for its next application.

It is clear, for the first application, that all the hypotheses of this lemma are satisfied.

In the end, we know that there exist Γ', some constraint sets C_i, some c_ϕ, c_σ, and a reduction

$$(\{Q\}, \emptyset, \emptyset) \xrightarrow{s'}_* (\mathcal{Q}, \phi_Q, \sigma_Q)$$

with $s =_\tau s'$, such that (among other conclusions)

- σ_P, σ_Q are ground, and there exist ground $\sigma'_P \supseteq \sigma_P$, $\sigma'_Q \supseteq \sigma_Q$ such that
  - $(\mathrm{dom}(\sigma'_P) \backslash \mathrm{dom}(\sigma_P)) \cap (\mathrm{vars}(\mathcal{P}) \cup \mathrm{vars}(\phi_P)) = \emptyset$,
  - $(\mathrm{dom}(\sigma'_Q) \backslash \mathrm{dom}(\sigma_Q)) \cap (\mathrm{vars}(\mathcal{Q}) \cup \mathrm{vars}(\phi_Q)) = \emptyset$, and
  - $\Gamma_{\mathcal{N},\mathcal{K}} \vdash \sigma'_P \sim \sigma'_Q : \Gamma_{\mathcal{X}} \to c_\sigma$,
  - for all $x \in \mathrm{dom}(\sigma'_P)$, $\sigma'_P(x)\downarrow = \sigma'_P(x)$, and similarly for σ'_Q,
- $c_\sigma \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$,
- $\Gamma' \vdash \phi_P \sim \phi_Q : \mathtt{LL} \to c_\phi$,
- $\mathrm{dom}(\phi_P) = \mathrm{dom}(\phi_Q)$,
- $\forall i, \Gamma' \vdash P_i \sim Q_i \to C_i$,
- for all $i \neq j$, the sets of bound variables in P_i and P_j (resp. Q_i and Q_j) are disjoint, and similarly for the bound names;
- $[\![(\cup_{\times i} C_i) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$ is consistent.

To prove the claim, it is then sufficient to show that $\phi_P\sigma_P$ and $\phi_Q\sigma_Q$ are statically equivalent.

Note that since $\sigma_P \subseteq \sigma'_P$ and $(\mathrm{dom}(\sigma'_P)\backslash\mathrm{dom}(\sigma_P)) \cap (\mathrm{vars}(\mathcal{P}) \cup \mathrm{vars}(\phi_P)) = \emptyset$, we have $\phi_P\sigma_P = \phi_P\sigma'_P$. Similarly $\phi_Q\sigma_Q = \phi_Q\sigma'_Q$. In addition we also know that $[\![c_\phi]\!]_{\sigma_P,\sigma_Q} = [\![c_\phi]\!]_{\sigma'_P,\sigma'_Q}$.

We have $\Gamma' \vdash \phi_P \sim \phi_Q : \mathtt{LL} \to c_\phi$ and $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma'_P \sim \sigma'_Q : \Gamma'_{\mathcal{X}} \to c_\sigma$. Hence, by Lemma A.22, there exists $c \subseteq [\![c_\phi]\!]_{\sigma'_P,\sigma'_Q} \cup c_\sigma$ (*i.e.* such that $c \subseteq [\![c_\phi]\!]_{\sigma'_P,\sigma'_Q}$) such that $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \phi_P\sigma'_P \sim \phi_Q\sigma'_Q : \mathtt{LL} \to c$.

We will now show that $(c, {\Gamma'}_{\mathcal{N},\mathcal{K}})$ is consistent.

Let $\Gamma'' \in \mathrm{branches}(\Gamma')$. By Lemma A.12, for all i, since $\Gamma' \vdash P_i \sim Q_i \to C_i$, there exists $(c_i, \Gamma'''_i) \in C_i$ such that $\Gamma'' \subseteq \Gamma'''_i$. The disjointness condition on the bound variables implies by Lemma A.8 that for all i, j, Γ'''_i and Γ'''_j are compatible. Thus $\cup_{\times i} C_i$ contains $(c', \Gamma''') \stackrel{\mathrm{def}}{=} (\bigcup_i c_i, \bigcup_i \Gamma'''_i)$. We have $\Gamma'' \subseteq \Gamma'''$. Therefore $[\![(\cup_{\times i} C_i) \cup_\forall c_\phi]\!]_{\sigma_P,\sigma_Q}$, which is consistent, contains $([\![c' \cup c_\phi]\!]_{\sigma_P,\sigma_Q}, \Gamma''')$. Therefore, as $c \subseteq [\![c_\phi]\!]_{\sigma_P,\sigma_Q}$, by Lemma A.11, (c, Γ''') is consistent. Since c is ground, it follows from the definition of consistency that $(c, {\Gamma'''}_{\mathcal{N},\mathcal{K}})$ is also consistent. Moreover, we know that $\Gamma'' \subseteq \Gamma'''$, and Γ'' is a branch of Γ'. It is then clear that ${\Gamma'}_{\mathcal{N},\mathcal{K}} \subseteq \Gamma'''$. Hence, by Lemma A.10, since ${\Gamma'}_{\mathcal{N},\mathcal{K}} \vdash \phi_P\sigma_P \sim \phi_Q\sigma_Q : \mathtt{LL} \to c$, we have $\Gamma''' \vdash \phi_P\sigma_P \sim \phi_Q\sigma_Q : \mathtt{LL} \to c$.

Hence, we have $\Gamma''' \vdash \phi_P\sigma_P \sim \phi_Q\sigma_Q : \mathtt{LL} \to c$ with $(c, {\Gamma'''}_{\mathcal{N},\mathcal{K}})$ consistent.
Moreover, $\phi_P\sigma_P$ and $\phi_Q\sigma_Q$ are ground (by well-formedness of the processes).
Therefore, by Lemma A.25, the frames $\phi_P\sigma_P$ and $\phi_Q\sigma_Q$ are statically equivalent.

□

Theorem A.2 (Typing implies trace equivalence)**.** *For all Γ containing only keys, for all P and Q, if*

$$\Gamma \vdash P \sim Q \to C$$

and C is consistent, then

$$P \approx_t Q.$$

Proof. Theorem A.1 proves that under these assumptions, $P \sqsubseteq_t Q$. This is sufficient to prove the theorem. Indeed, it is clear from the typing rules for processes and terms that

$$\Gamma \vdash P \sim Q \to C \Leftrightarrow \Gamma' \vdash Q \sim P \to C'$$

where C' is the constraint obtained from C by swapping the left and right hand sides of all of its elements, and Γ' is the environment obtained from Γ by swapping the left and right types in all refinement types, as well as swapping all pairs of keys in its domain. Clearly from the definition of consistency, C is consistent if and only if C' is. Therefore, by symmetry, proving that the assumptions imply $P \sqsubseteq_t Q$ also proves that they imply $Q \sqsubseteq_t P$, and thus $P \approx_t Q$. □

A.2 Typing replicated processes

In this section, we prove the soundness result for replicated processes. We recall that a table displaying the correspondence between the numbering of the Lemmas and Theorems in Chapter 2 and here is provided at the beginning of Appendix A.

In this section, as well as the following ones, without loss of generality we assume, for each infinite nonce type $\tau_m^{l,\infty}$ appearing in the processes we consider, that $\mathcal{N}$ contains an infinite number of fresh names which we will denote by $\{m_i \mid i \in \mathbb{N}\}$; such that the m_i do not appear in

the processes or environments considered. We will denote by $\mathcal{N}_0$ the set of unindexed names and by $\mathcal{N}_i$ the set of indexed names. We similarly assume that for all the variables x appearing in the processes, the set $\mathcal{X}$ of all variables also contains variables $\{x_i \mid i \in \mathbb{N}\}$. We denote $\mathcal{X}_0$ the set of unindexed variables, and $\mathcal{X}_i$ the set of indexed variables. Finally, we assume for all key k declared in the processes with type $\mathrm{seskey}^{l,\infty}(T)$ that the set $\mathcal{BK}$ contains keys $\{k_i \mid i \in \mathbb{N}\}$.

Lemma A.27 (Typing terms with replicated names)**.** *For all* Γ, M, N, T *and* c, *if*

$$\Gamma \vdash M \sim N : T \to c$$

then for all $i, n \in \mathbb{N}$ *such that* $1 \leq i \leq n$, *for all* $\Gamma' \in \mathrm{branches}([\,\Gamma\,]_i^n)$,

$$\Gamma' \vdash [\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma : [\,T\,]^n \to [\,c\,]_i^\Gamma$$

Proof. Let Γ, M, N, T, c be such as assumed in the statement of the lemma. Let $i, n \in \mathbb{N}$ such that $1 \leq i \leq n$. Let $\Gamma' \in \mathrm{branches}([\,\Gamma\,]_i^n)$.

We prove this property by induction on the proof Π of

$$\Gamma \vdash M \sim N : T \to c.$$

There are several possible cases for the last rule applied in Π.

- TNonce: then $M = m$ and $N = p$ for some $m, p \in \mathcal{N}$, $T = l$ for some $l \in \{\mathtt{HH}, \mathtt{HL}\}$, and

$$\Pi = \frac{\Gamma(m) = \tau_m^{l,a} \qquad \Gamma(p) = \tau_p^{l,a}}{\Gamma \vdash m \sim p : l \to \emptyset}.$$

  It is clear from the definition of $[\,\Gamma\,]_i^n$ that $[\,\Gamma\,]_i^n([\,m\,]_i^\Gamma) = \tau^{l,1}_{[\,m\,]_i^\Gamma}$, and that $[\,\Gamma\,]_i^n([\,p\,]_i^\Gamma) = \tau^{l,1}_{[\,p\,]_i^\Gamma}$. Hence, $\Gamma'([\,m\,]_i^\Gamma) = \tau^{l,1}_{[\,m\,]_i^\Gamma}$ and $\Gamma'([\,p\,]_i^\Gamma) = \tau^{l,1}_{[\,p\,]_i^\Gamma}$. Then, by rule TNonce, we have $\Gamma' \vdash [\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma : l \to \emptyset$ and the claim holds.

- TNonceL, TCstFN, TKey, TPubKeyL, TVKeyL, THigh, TLR[1]: Similarly to the TNonce case, the claim follows directly from the definition of $[\,\Gamma\,]_i^n$, $[\,M\,]_i^\Gamma$, $[\,N\,]_i^\Gamma$, $[\,T\,]^n$ and $[\,c\,]_i^\Gamma$ in these cases.

- TEncH: then $T = \mathtt{LL}$ and there exist T', k, c' such that

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash M \sim N : (T')_{T''} \to c'} \qquad T'' <: \mathrm{key}^{\mathtt{HH}}(T')}{\Gamma \vdash M \sim N : \mathtt{LL} \to c = c' \cup \{M \sim N\}}.$$

  By applying the induction hypothesis to Π', since $[\,(T')_{T''}\,]^n = ([\,T'\,]^n)_{[\,T''\,]^n}$, there exists a proof Π'' of $\Gamma' \vdash [\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma : ([\,T'\,]^n)_{[\,T''\,]^n} \to [\,c'\,]_i^\Gamma$

  In addition it is clear by definition of $[\,\cdot\,]^n$ and Lemma A.1 that since $T'' <: \mathrm{key}^{\mathtt{HH}}(T')$ we have $[\,T''\,]^n <: \mathrm{key}^{\mathtt{HH}}([\,T'\,]^n)$.

  Therefore by rule TEncH, we have

$$\Gamma' \vdash [\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma : \mathtt{LL} \to [\,c'\,]_i^\Gamma \cup \{[\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma\} = [\,c\,]_i^\Gamma$$

- TPubKey, TVKey, TEnc, TEncL, TAenc, TAencH, TAencL,TSignH, TSignL, as well as TPair, TOr,THash, THashL: Similarly to the TEncH case, the claim is proved directly by applying the induction hypothesis to the type judgement appearing in the conditions of the last rule in these cases.

- TVar: then $M = N = x$ for some $x \in \mathcal{X}$, and

$$\Pi = \frac{\Gamma(x) = T}{\Gamma \vdash x \sim x : T \to \emptyset}.$$

  We have $[\, M \,]_i^\Gamma = [\, N \,]_i^\Gamma = x_i$.
  Since $\Gamma' \in \text{branches}([\, \Gamma \,]_i^n)$, we have $\Gamma'(x_i) \in \text{branches}([\, T \,]^n)$.
  Hence by rule TVar, $\Gamma' \vdash x_i \sim x_i : \Gamma'(x_i) \to \emptyset$. Therefore, by rule TOr, we have

$$\Gamma' \vdash x_i \sim x_i : [\, T \,]^n \to \emptyset$$

  which proves the claim.

- TLR' (the TLRL' case is similar): then there exist m, p, l such that $T = l$, and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash M \sim N : [\![\tau_m^{l,a} \,;\, \tau_p^{l,a}]\!] \to c}}{\Gamma \vdash M \sim N : l \to c}.$$

  Let us distinguish the case where a is 1 from the case where a is ∞.
  If a is 1: by applying the induction hypothesis to Π', since $\left[\, [\![\tau_m^{l,a} \,;\, \tau_p^{l,a}]\!] \,\right]^n = [\![\tau_m^{l,1} \,;\, \tau_p^{l,1}]\!]$, we have

$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : [\![\tau_m^{l,a} \,;\, \tau_p^{l,a}]\!] \to [\, c \,]_i^\Gamma.$$

  Thus by rule TLR', we have

$$[\, \Gamma \,]_i^n \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : l \to [\, c \,]_i^\Gamma.$$

  If a is ∞: by applying the induction hypothesis to Π', since

$$\left[\, [\![\tau_m^{l,a} \,;\, \tau_p^{l,a}]\!] \,\right]^n = \bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1} \,;\, \tau_{p_j}^{l,1}]\!],$$

  we have

$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : \bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1} \,;\, \tau_{p_j}^{l,1}]\!] \to [\, c \,]_i^\Gamma.$$

  Thus, by Lemma A.5, there exists $j \in [\![1, n]\!]$ and a proof Π'' of

$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : [\![\tau_{m_j}^{l,1} \,;\, \tau_{p_j}^{l,1}]\!] \to [\, c \,]_i^\Gamma.$$

  Thus, by rule TLR',

$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : l \to [\, c \,]_i^\Gamma,$$

  which proves the claim.

- TLRVAR: this case is similar to the TLR' case, but only the case where a is 1 is possible.
- TSUB: then there exists $T' <: T$ such that

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash M \sim N : T' \to c} \qquad T' <: T}{\Gamma \vdash M \sim N : T \to c}.$$

  By applying the induction hypothesis to Π', we have

$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : [\, T' \,]^n \to [\, c \,]_i^\Gamma.$$

  Since it is clear by induction on the subtyping rules that $T' <: T$ implies that $[\, T' \,]^n <: [\, T \,]^n$, the TSUB rule can be applied and proves the claim.

- TLR$^\infty$: then $M = m$, $N = p$, $c = \emptyset$, and $T = [\![\tau_m^{l,\infty} ; \tau_p^{l',\infty}]\!]$ for some $m, p \in \mathcal{N}$, $c = \emptyset$, and

$$\Pi = \frac{\Gamma(m) = \tau_m^{l,\infty} \qquad \Gamma(p) = \tau_p^{l',\infty}}{\Gamma \vdash m \sim p : [\![\tau_m^{l,\infty} ; \tau_p^{l',\infty}]\!] \to \emptyset}.$$

  We have by definition $[\, M \,]_i^\Gamma = m_i$ and $[\, N \,]_i^\Gamma = p_i$, and $[\, \Gamma \,]_i^n(m_i) = \tau_{m_i}^{l,1}$, and $[\, \Gamma \,]_i^n(p_i) = \tau_{p_i}^{l',1}$. Thus $\Gamma'(m_i) = \tau_{m_i}^{l,1}$, and $\Gamma'(p_i) = \tau_{p_i}^{l',1}$. Hence by rule TLR^1, we have $\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : [\![\tau_{m_i}^{l,1} ; \tau_{p_i}^{l',1}]\!] \to \emptyset$.

  In addition, $\Big[\, [\![\tau_m^{l,\infty} ; \tau_p^{l',\infty}]\!] \,\Big]^n = \bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1} ; \tau_{p_j}^{l',1}]\!]$. Therefore, by applying rule TOR, we have

$$\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : \Big[\, [\![\tau_m^{l,\infty} ; \tau_p^{l',\infty}]\!] \,\Big]^n \to \emptyset$$

  which proves the claim.

□

Lemma A.28 (Typing destructors with replicated names)**.** *For all* Γ, t, t', T, *if*

$$\Gamma \vdash t \sim t' : T$$

then for all $i, n \in \mathbb{N}$ *such that* $1 \le i \le n$,

$$[\, \Gamma \,]_i^n \vdash [\, t \,]_i^\Gamma \sim [\, t' \,]_i^\Gamma : [\, T \,]^n$$

Proof. Immediate by examining the typing rules for destructors. □

Lemma A.29 (Branches and expansion)**.**

- *For all* T,

$$\bigcup_{T' \in \mathrm{branches}(T)} \mathrm{branches}([\, T' \,]^n) = \mathrm{branches}([\, T \,]^n)$$

- *For all* Γ *for all* $i, n \in \mathbb{N}$,

$$\bigcup_{\Gamma' \in \mathrm{branches}(\Gamma)} \mathrm{branches}([\, \Gamma' \,]_i^n) = \mathrm{branches}([\, \Gamma \,]_i^n)$$

Proof. The first point is proved by induction on T.

If $T = T' \vee T''$ for some T', T'', then

$$\begin{array}{rcl}\mathrm{branches}([\,T\,]^n) & = & \mathrm{branches}([\,T'\,]^n) \cup \mathrm{branches}([\,T''\,]^n) \\ & = & (\bigcup_{T'''\in\mathrm{branches}(T')} \mathrm{branches}([\,T'''\,]^n)) \cup (\bigcup_{T'''\in\mathrm{branches}(T'')} \mathrm{branches}([\,T'''\,]^n))\end{array}$$

by the induction hypothesis. Since $\mathrm{branches}(T) = \mathrm{branches}(T') \cup \mathrm{branches}(T'')$, this proves the claim.

Otherwise, $\mathrm{branches}(T) = \{T\}$ and the claim trivially holds.

The second point directly follows from the first point, using the definition of $[\,\Gamma\,]^n_i$. □

Lemma A.30 (Typing processes in all branches)**.** *For all P, Q, Γ, $\{C_{\Gamma'}\}_{\Gamma'\in\mathrm{branches}(\Gamma)}$, if*

$$\forall \Gamma' \in \mathrm{branches}(\Gamma). \quad \Gamma' \vdash P \sim Q \to C_{\Gamma'}$$

then

$$\Gamma \vdash P \sim Q \to \bigcup_{\Gamma'\in\mathrm{branches}(\Gamma)} C_{\Gamma'}.$$

Consequently if for some C, $C_{\Gamma'} \subseteq C$ for all Γ', then there exists $C' \subseteq C$ such that

$$\Gamma \vdash P \sim Q \to C'.$$

Proof. The first point is easily proved by successive applications of rule POR.

The second point is a direct consequence of the first point. □

Lemma A.31 (Expansion and union)**.**

- *For all constraint sets C, C', such that $\forall (c,\Gamma) \in C \cup C'$. $\mathrm{branches}(\Gamma) = \{\Gamma\}$,* i.e. *such that Γ does not contain union types, and $\mathrm{names}(c) \subseteq \mathrm{dom}(\Gamma) \cup \mathcal{FN}$, and Γ only nonce types with names from $\mathcal{N}_0$ (*i.e. *unindexed names), we have*
$$[\,C \cup_\times C'\,]^n_i = [\,C\,]^n_i \cup_\times [\,C'\,]^n_i$$
- *For all C, c, Γ, such that $\mathrm{names}(c) \subseteq \mathrm{dom}(\Gamma)$ and $\forall (c,\Gamma') \in C$. $\Gamma_{\mathcal{N},\mathcal{K}} \subseteq \Gamma'$, we have*
$$[\,C \cup_\forall c\,]^n_i = [\,C\,]^n_i \cup_\forall [\,c\,]^\Gamma_i$$

Proof. The first point follows from the definition of $[\,\cdot\,]^n_i$ and $\cup_\times$. Indeed, if C, C' are as assumed in the claim, we have:

$$\begin{array}{rcl}[\,C\cup_\times C'\,]^n_i & = & \{([\,c\,]^\Gamma_i, \Gamma') | \exists \Gamma.\ (c,\Gamma) \in C \cup_\times C' \ \wedge\ \Gamma' \in \mathrm{branches}([\,\Gamma\,]^n_i)\} \\ & = & \{([\,c_1 \cup c_2\,]^{\Gamma_1\cup\Gamma_2}_i, \Gamma') | \exists \Gamma_1\Gamma_2.\ (c_1,\Gamma_1) \in C \ \wedge\ (c_2,\Gamma_2) \in C' \ \wedge\ \Gamma_1,\Gamma_2 \text{ are compatible} \wedge \\ & & \quad \Gamma' \in \mathrm{branches}([\,\Gamma_1 \cup \Gamma_2\,]^n_i)\} \\ & = & \{([\,c_1\,]^{\Gamma_1}_i \cup [\,c_2\,]^{\Gamma_2}_i, \Gamma') | \exists \Gamma_1\Gamma_2.\ (c_1,\Gamma_1) \in C \ \wedge\ (c_2,\Gamma_2) \in C' \ \wedge\ \Gamma_1,\Gamma_2 \text{ are compatible} \wedge \\ & & \quad \Gamma' \in \mathrm{branches}([\,\Gamma_1\,]^n_i \cup [\,\Gamma_2\,]^n_i)\} \\ & = & \{([\,c_1\,]^{\Gamma_1}_i \cup [\,c_2\,]^{\Gamma_2}_i, \Gamma\cup\Gamma') | \exists \Gamma_1\Gamma_2.\ (c_1,\Gamma_1) \in C \ \wedge\ (c_2,\Gamma_2) \in C' \ \wedge\ \Gamma_1,\Gamma_2 \text{ are compatible} \wedge \\ & & \quad \Gamma \in \mathrm{branches}([\,\Gamma_1\,]^n_i) \ \wedge\ \Gamma' \in \mathrm{branches}([\,\Gamma_2\,]^n_i) \ \wedge\ \Gamma,\Gamma' \text{ are compatible}\}\end{array}$$

The last step is proved by directly showing both inclusions.

On the other hand we have:

$$\begin{array}{rcl}[\,C\,]^n_i \cup_\times [\,C'\,]^n_i & = & \{(c\cup c', \Gamma\cup\Gamma') | (c,\Gamma) \in [\,C\,]^n_i \ \wedge\ (c',\Gamma') \in [\,C'\,]^n_i \ \wedge\ \Gamma,\Gamma' \text{ are compatible}\} \\ & = & \{([\,c_1\,]^{\Gamma_1}_i \cup [\,c_2\,]^{\Gamma_2}_i, \Gamma\cup\Gamma') | \exists \Gamma_1\Gamma_2.\ (c_1,\Gamma_1) \in C \ \wedge\ (c_2,\Gamma_2) \in C' \ \wedge \\ & & \quad \Gamma \in \mathrm{branches}([\,\Gamma_1\,]^n_i) \ \wedge\ \Gamma' \in \mathrm{branches}([\,\Gamma_2\,]^n_i) \ \wedge\ \Gamma,\Gamma' \text{ are compatible}\} \\ & = & \{([\,c_1\,]^{\Gamma_1}_i \cup [\,c_2\,]^{\Gamma_2}_i, \Gamma\cup\Gamma') | \exists \Gamma_1\Gamma_2.\ (c_1,\Gamma_1) \in C \ \wedge\ (c_2,\Gamma_2) \in C' \ \wedge \\ & & \quad \Gamma \in \mathrm{branches}([\,\Gamma_1\,]^n_i) \ \wedge\ \Gamma' \in \mathrm{branches}([\,\Gamma_2\,]^n_i) \ \wedge \\ & & \quad \Gamma,\Gamma' \text{ are compatible} \ \wedge\ \Gamma_1,\Gamma_2 \text{ are compatible}\}\end{array}$$

This last step comes from the fact that if $(c_1, \Gamma_1) \in C$ and $(c_2, \Gamma_2) \in C'$, then by assumption Γ_1 and Γ_2 do not contain union types. This implies that if $\Gamma \in \text{branches}([\,\Gamma_1\,]_i^n)$ and $\Gamma' \in \text{branches}([\,\Gamma_2\,]_i^n)$ are compatible, then Γ_1 and Γ_2 are compatible. Indeed, let $x \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2)$. Hence $x_i \in \text{dom}(\Gamma) \cap \text{dom}(\Gamma')$, and since they are compatible, $\Gamma(x_i) = \Gamma'(x_i)$. That is to say that there exists $T \in \text{branches}([\,\Gamma_1(x)\,]^n) \cap \text{branches}([\,\Gamma_2(x)\,]^n)$.

If $\Gamma_1(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$ (for some m, p, l, l'), then $[\,\Gamma_1(x)\,]^n = \bigvee_{1\leq j\leq n}[\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]$, and thus there exists $j \in [\![1, n]\!]$ such that $T = [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]$. Hence, $[\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!] \in \text{branches}([\,\Gamma_2(x)\,]^n)$. Because of the definition of $[\,\cdot\,]^n$, and since $\Gamma_2(x)$ is not a union type (by assumption), this implies that $\Gamma_2(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$, and therefore $\Gamma_1(x) = \Gamma_2(x)$.

If $\Gamma_1(x)$ is not of the form $[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$ (for some m, p, l, l'), then neither is $\Gamma_2(x)$ (by contraposition, following the same reasoning as in the previous case). $\Gamma_1(x)$ and $\Gamma_2(x)$ are not of the form $T' \vee T''$ either, by assumption. Therefore, neither $[\,\Gamma_1(x)\,]^n$ nor $[\,\Gamma_2(x)\,]^n$ are union types (from the definition of $[\,\cdot\,]^n$). This implies that $T = [\,\Gamma_1(x)\,]^n = [\,\Gamma_2(x)\,]^n$, which implies $\Gamma_1(x) = \Gamma_2(x)$.

In both cases $\Gamma_1(x) = \Gamma_2(x)$, and Γ_1, Γ_2 are therefore compatible.

Hence $[\,C\,]_i^n \cup_\times [\,C'\,]_i^n = [\,C \cup_\times C'\,]_i^n$, which proves the claim.

The second point directly follows from the definition of $[\,\cdot\,]_i^n$ and $\cup_\forall$. Indeed, for all C, c, Γ satisfying the assumptions, we have:

$$\begin{aligned}
[\,C \cup_\forall c\,]_i^n &= \{([\,c'\,]_i^{\Gamma'}, \Gamma'') | \exists \Gamma'.\ (c', \Gamma') \in C \cup_\forall c \ \wedge\ \Gamma'' \in \text{branches}([\,\Gamma'\,]_i^n)\} \\
&= \{([\,c'' \cup c\,]_i^{\Gamma'}, \Gamma'') | \exists \Gamma'.\ (c'', \Gamma') \in C \ \wedge\ \Gamma'' \in \text{branches}([\,\Gamma'\,]_i^n)\} \\
&= \{([\,c''\,]_i^{\Gamma'} \cup [\,c\,]_i^{\Gamma'}, \Gamma'') | \exists \Gamma'.\ (c'', \Gamma') \in C \ \wedge\ \Gamma'' \in \text{branches}([\,\Gamma'\,]_i^n)\} \\
&= \{([\,c''\,]_i^{\Gamma'} \cup [\,c\,]_i^{\Gamma}, \Gamma'') | \exists \Gamma'.\ (c'', \Gamma') \in C \ \wedge\ \Gamma'' \in \text{branches}([\,\Gamma'\,]_i^n)\} \\
&\qquad \text{(since } \Gamma, \Gamma' \text{ give the same types to names and keys)} \\
&= \{([\,c'\,]_i^{\Gamma'}, \Gamma'') | \exists \Gamma'.\ (c', \Gamma') \in C \ \wedge\ \Gamma'' \in \text{branches}([\,\Gamma'\,]_i^n)\} \cup_\forall [\,c\,]_i^\Gamma \\
&= [\,C\,]_i^n \cup_\forall [\,c\,]_i^\Gamma
\end{aligned}$$

□

Theorem A.3 (Typing processes with expanded types)**.** *For all* Γ*,* P*,* Q *and* C*, if*

$$\Gamma \vdash P \sim Q \to C$$

then for all $i, n \in \mathbb{N}$ *such that* $1 \leq i \leq n$*, there exists* $C' \subseteq [\,C\,]_i^n$ *such that*

$$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C'$$

Proof. We prove this theorem by induction on the derivation Π of $\Gamma \vdash P \sim Q \to C$. We distinguish several cases for the last rule applied in this derivation.

- PZero: then $P = Q = [\,P\,]_i^\Gamma = [\,Q\,]_i^\Gamma = 0$, and $C = \{(\emptyset, \Gamma)\}$. Hence

  $$[\,C\,]_i^n = \{(\emptyset, \Gamma') \mid \Gamma' \in \text{branches}([\,\Gamma\,]_i^n)\}$$

  Thus, by applying rule POr as many times as necessary to split $[\,\Gamma\,]_i^n$ into all of its branches, followed by rule PZero, we have $[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to [\,C\,]_i^n$.

- POUT: then $P = \mathtt{out}(M).P'$, $Q = \mathtt{out}(N).Q'$ for some messages M, N and some processes P', Q', and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash P' \sim Q' \to C'} \qquad \dfrac{\Pi''}{\Gamma \vdash M \sim N : \mathtt{LL} \to c}}{\Gamma \vdash P \sim Q \to C = C' \cup_\forall c}.$$

  By applying the induction hypothesis to Π', there exists $C'' \subseteq [\, C' \,]_i^n$ and a proof Π''' of $[\, \Gamma \,]_i^n \vdash [\, P' \,]_i^\Gamma \sim [\, Q' \,]_i^\Gamma \to C''$.

  Hence, by Lemma A.7, for all $\Gamma' \in \mathrm{branches}([\, \Gamma \,]_i^n)$, there exist $C_{\Gamma'} \subseteq C''$ and a proof $\Pi_{\Gamma'}$ of $\Gamma' \vdash [\, P' \,]_i^\Gamma \sim [\, Q' \,]_i^\Gamma \to C_{\Gamma'}$.

  Moreover, by Lemma A.27, for all $\Gamma' \in \mathrm{branches}([\, \Gamma \,]_i^n)$, there exists a proof $\Pi'_{\Gamma'}$ of $\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : \mathtt{LL} \to [\, c \,]_i^\Gamma$.

  In addition, $[\, P \,]_i^\Gamma = [\, \mathtt{out}(M).P' \,]_i^\Gamma = \mathtt{out}([\, M \,]_i^\Gamma).[\, P' \,]_i^\Gamma$. Similarly, we have $[\, Q \,]_i^\Gamma = \mathtt{out}([\, N \,]_i^\Gamma).[\, Q' \,]_i^\Gamma$. Therefore, using $\Pi_{\Gamma'}$, $\Pi'_{\Gamma'}$ and rule POUT, we have for all $\Gamma' \in \mathrm{branches}([\, \Gamma \,]_i^n)$ that $\Gamma' \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C_{\Gamma'} \cup_\forall [\, c \,]_i^\Gamma \subseteq C'' \cup_\forall [\, c \,]_i^\Gamma$.

  Thus by Lemma A.30, there exists $C_1 \subseteq C'' \cup_\forall [\, c \,]_i^\Gamma$ such that

$$[\, \Gamma \,]_i^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C_1.$$

  Finally, $[\, C \,]_i^n = [\, C' \cup_\forall c \,]_i^n = [\, C' \,]_i^n \cup_\forall [\, c \,]_i^\Gamma$ (by Lemma A.31, whose conditions are satisfied, by Lemma A.12). Hence $C'' \cup_\forall [\, c \,]_i^\Gamma \subseteq [\, C \,]_i^n$, which proves the claim.

- PIN: then $P = \mathtt{in}(x).P'$, $Q = \mathtt{in}(x).Q'$ for some variable x and some processes P', Q', and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma, x : \mathtt{LL} \vdash P' \sim Q' \to C}}{\Gamma \vdash P \sim Q \to C}.$$

  Since

$$[\, \Gamma, x : \mathtt{LL} \,]_i^n = [\, \Gamma \,]_i^n, x_i : [\, \mathtt{LL} \,]^n = [\, \Gamma \,]_i^n, x_i : \mathtt{LL},$$

  by applying the induction hypothesis to Π', there exists $C' \subseteq [\, C \,]_i^n$ and a proof Π'' of $[\, \Gamma \,]_i^n, x : \mathtt{LL} \vdash [\, P' \,]_i^{\Gamma, x:\mathtt{LL}} \sim [\, Q' \,]_i^{\Gamma, x:\mathtt{LL}} \to C'$.

  In addition, $[\, P \,]_i^\Gamma = [\, \mathtt{in}(x).P' \,]_i^\Gamma = \mathtt{in}(x_i).[\, P' \,]_i^\Gamma = \mathtt{in}(x_i).[\, P' \,]_i^{\Gamma, x:\mathtt{LL}}$. Similarly, we have $[\, Q \,]_i^\Gamma = \mathtt{in}(x_i).[\, Q' \,]_i^{\Gamma, x:\mathtt{LL}}$.

  Therefore, using Π'' and rule PIN, we have $[\, \Gamma \,]_i^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C' \subseteq [\, C \,]_i^n$.

- PNEW: then $P = \mathtt{new}\ m : \tau_m^{l,a}.P'$, $Q = \mathtt{new}\ m : \tau_m^{l,a}.Q'$ for some m, l, a and some processes P', Q', and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma, m : \tau_m^{l,a} \vdash P' \sim Q' \to C}}{\Gamma \vdash P \sim Q \to C}.$$

  – If $a = 1$:
  Since

$$\left[\, \Gamma, m : \tau_m^{l,1} \,\right]_i^n = [\, \Gamma \,]_i^n, m : \tau_m^{l,1},$$

by applying the induction hypothesis to Π', there exists $C' \subseteq [\, C \,]_i^n$ and a proof Π'' of $[\, \Gamma \,]_i^n, m : \tau_m^{l,1} \vdash [\, P' \,]_i^{\Gamma, m:\tau_m^{l,1}} \sim [\, Q' \,]_i^{\Gamma, m:\tau_m^{l,1}} \to C'$.
In addition $[\, P \,]_i^\Gamma = [\, \mathtt{new}\ m.P' \,]_i^\Gamma$, which is to say that $[\, P \,]_i^\Gamma = \mathtt{new}\ m : \tau_m^{l,1}.[\, P' \,]_i^\Gamma = \mathtt{new}\ m : \tau_m^{l,1}.[\, P' \,]_i^{\Gamma, m:\tau_m^{l,1}}$; and similarly for Q. Therefore, using Π'' and rule PNew, we have $[\, \Gamma \,]_i^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C' \subseteq [\, C \,]_i^n$.

– If $a = \infty$: Since

$$\left[\, \Gamma, m : \tau_m^{l,\infty} \,\right]_i^n = [\, \Gamma \,]_i^n, m_i : \tau_{m_i}^{l,1},$$

by applying the induction hypothesis to Π', there exists $C' \subseteq [\, C \,]_i^n$ and a proof Π'' of $[\, \Gamma \,]_i^n, m_i : \tau_{m_i}^{l,1} \vdash [\, P' \,]_i^{\Gamma, m:\tau_m^{l,\infty}} \sim [\, Q' \,]_i^{\Gamma, m:\tau_m^{l,\infty}} \to C'$.
In addition $[\, P \,]_i^\Gamma = [\, \mathtt{new}\ m.P' \,]_i^\Gamma = \mathtt{new}\ m_i : \tau_{m_i}^{l,1}.[\, P'[m_i/m] \,]_i^\Gamma$, which is to say that $[\, P \,]_i^\Gamma = \mathtt{new}\ m_i : \tau_{m_i}^{l,1}.[\, P' \,]_i^{\Gamma, m:\tau_m^{l,\infty}}$; and similarly for Q. Therefore, using Π'' and rule PNew, we have $[\, \Gamma \,]_i^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C' \subseteq [\, C \,]_i^n$.

- PNewKey: this case is similar to the PNew case.

- PPar: then $P = P' \mid P''$, $Q = Q' \mid Q''$ for some processes P', Q', P'', Q'', and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash P' \sim Q' \to C'} \qquad \dfrac{\Pi''}{\Gamma \vdash P'' \sim Q'' \to C''}}{\Gamma \vdash P \sim Q \to C = C' \cup_\times C''}.$$

By applying the induction hypothesis to Π', there exists $C''' \subseteq [\, C' \,]_i^n$ and a proof Π''' of $[\, \Gamma \,]_i^n \vdash [\, P' \,]_i^\Gamma \sim [\, Q' \,]_i^\Gamma \to C'''$. Similarly, by applying the induction hypothesis to Π'', there exists $C'''' \subseteq [\, C'' \,]_i^n$ and a proof Π'''' of $[\, \Gamma \,]_i^n \vdash [\, P'' \,]_i^\Gamma \sim [\, Q'' \,]_i^\Gamma \to C''''$.

In addition, $[\, P \,]_i^\Gamma = [\, P' \mid P'' \,]_i^\Gamma = [\, P' \,]_i^\Gamma \mid [\, P'' \,]_i^\Gamma$. Similarly, $[\, Q \,]_i^\Gamma = [\, Q' \mid Q'' \,]_i^\Gamma = [\, Q' \,]_i^\Gamma \mid [\, Q'' \,]_i^\Gamma$. Finally, $[\, C \,]_i^n = [\, C' \cup_\times C'' \,]_i^n = [\, C' \,]_i^n \cup_\times [\, C'' \,]_i^n$, by Lemma A.31 (using Lemma A.9 to ensure the condition that the environments do not contain union types).

Therefore, using Π''', Π'''' and rule PPar, we have $[\, \Gamma \,]_i^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C''' \cup_\times C'''' \subseteq [\, C \,]_i^n$.

- POr: then $\Gamma = \Gamma', x : T \vee T'$ for some Γ', some $x \in \mathcal{X}$ and some types T, T', and

$$\Pi = \frac{\dfrac{\Pi_T}{\Gamma', x : T \vdash P \sim Q \to C'} \qquad \dfrac{\Pi_{T'}}{\Gamma', x : T' \vdash P \sim Q \to C''}}{\Gamma \vdash P \sim Q \to C = C' \cup C''}.$$

By applying the induction hypothesis to Π_T, there exist $C_1 \subseteq [\, C' \,]_i^n$ and a proof Π_1 of $[\, \Gamma' \,]_i^n, x_i : [\, T \,]^n \vdash [\, P \,]_i^{\Gamma', x:T} \sim [\, Q \,]_i^{\Gamma', x:T} \to C_1$. Similarly with $\Pi_{T'}$, there exist $C_2 \subseteq [\, C'' \,]_i^n$ and a proof Π_2 of $[\, \Gamma' \,]_i^n, x_i : [\, T' \,]^n \vdash [\, P \,]_i^{\Gamma', x:T'} \sim [\, Q \,]_i^{\Gamma', x:T'} \to C_2$.

In addition $[\, P \,]_i^\Gamma = [\, P \,]_i^{\Gamma', x:T} = [\, P \,]_i^{\Gamma', x:T'}$, and similarly for Q.

Thus by rule POr, we have

$$[\, \Gamma' \,]_i^n, x_i : [\, T \,]^n \vee [\, T' \,]^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C_1 \cup C_2 \subseteq [\, C' \,]_i^n \cup [\, C'' \,]_i^n = [\, C \,]_i^n.$$

Since $[\, \Gamma \,]_i^n = [\, \Gamma', x : T \vee T' \,]_i^n = [\, \Gamma' \,]_i^n, x_i : [\, T \,]^n \vee [\, T' \,]^n$, this proves the claim in this case.

- PLET: then $P = \texttt{let}\ x = t\ \texttt{in}\ P'\ \texttt{else}\ P''$, $Q = \texttt{let}\ x = t'\ \texttt{in}\ Q'\ \texttt{else}\ Q''$ for some variable x and some processes P', Q', P'', Q'', and

$$\Pi = \frac{\dfrac{\Pi_d}{\Gamma \vdash t \sim t' : T} \qquad \dfrac{\Pi'}{\Gamma, x : T \vdash P' \sim Q' \to C'} \qquad \dfrac{\Pi''}{\Gamma \vdash P'' \sim Q'' \to C''}}{\Gamma \vdash P \sim Q \to C = C' \cup C''}.$$

  Since

$$[\,\Gamma, x : T\,]^n_i = [\,\Gamma\,]^n_i, x_i : [\,T\,]^n,$$

  by applying the induction hypothesis to Π', there exist $C''' \subseteq [\,C'\,]^n_i$ and a proof Π''' of $[\,\Gamma\,]^n_i, x_i : [\,T\,]^n \vdash [\,P'\,]^{\Gamma,x:T}_i \sim [\,Q'\,]^{\Gamma,x:T}_i \to C'''$. Similarly, there exist $C'''' \subseteq [\,C''\,]^n_i$ and a proof Π'''' of $[\,\Gamma\,]^n_i \vdash [\,P''\,]^{\Gamma}_i \sim [\,Q''\,]^{\Gamma}_i \to C''''$.

  By Lemma A.28 applied to Π_d, we also have

$$[\,\Gamma\,]^n_i \vdash [\,t\,]^{\Gamma}_i \sim [\,t'\,]^{\Gamma}_i : [\,T\,]^n$$

  In addition, $[\,P\,]^{\Gamma}_i = [\,\texttt{let}\ x = t\ \texttt{in}\ P'\ \texttt{else}\ P''\,]^{\Gamma}_i = \texttt{let}\ x_i = [\,t\,]^{\Gamma}_i\ \texttt{in}\ [\,P'\,]^{\Gamma}_i\ \texttt{else}\ [\,P''\,]^{\Gamma}_i = \texttt{let}\ x_i = [\,t\,]^{\Gamma}_i\ \texttt{in}\ [\,P'\,]^{\Gamma,x:T}_i\ \texttt{else}\ [\,P''\,]^{\Gamma}_i$. Similarly, $[\,Q\,]^{\Gamma}_i = \texttt{let}\ x_i = [\,t'\,]^{\Gamma}_i\ \texttt{in}\ [\,Q'\,]^{\Gamma,x:T}_i\ \texttt{else}\ [\,Q''\,]^{\Gamma}_i$.

  Therefore, using Π'' and rule PLET, we have $[\,\Gamma\,]^n_i \vdash [\,P\,]^{\Gamma}_i \sim [\,Q\,]^{\Gamma}_i \to C''' \cup C'''' \subseteq [\,C\,]^n_i$.

- PLETDEC: then $P = \texttt{let}\ x = \texttt{dec}(y, k_1)\ \texttt{in}\ P'\ \texttt{else}\ P''$, and $Q = \texttt{let}\ x = \texttt{dec}(y, k_2)\ \texttt{in}\ Q'\ \texttt{else}\ Q''$ for some variable x, some keys k_1, k_2, and some processes P', Q', P'', Q'', and

$$\Pi = \frac{\begin{array}{c}\Gamma(y) = \texttt{LL} \qquad \Gamma(k_1, k_2) <: \text{key}^{\texttt{HH}}(T) \qquad \dfrac{\Pi'}{\Gamma, x : T \vdash P' \sim Q' \to C'} \qquad \dfrac{\Pi''}{\Gamma \vdash P'' \sim Q'' \to C''} \\ (\forall T'.\forall k_3 \neq k_2.\ \Gamma(k_1, k_3) <: \text{key}^{\texttt{HH}}(T') \Rightarrow \dfrac{\Pi^{1,k_3}}{\Gamma, x : T' \vdash P' \sim Q'' \to C_{k_3}}) \\ (\forall T'.\forall k_3 \neq k_2.\ \Gamma(k_3, k_2) <: \text{key}^{\texttt{HH}}(T') \Rightarrow \dfrac{\Pi^{2,k_3}}{\Gamma, x : T' \vdash P'' \sim Q' \to C'_{k_3}})\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{let}\ x = \texttt{dec}(y, k_1)\ \texttt{in}\ P'\ \texttt{else}\ P'' \sim \texttt{let}\ x = \texttt{dec}(y, k_2)\ \texttt{in}\ Q'\ \texttt{else}\ Q'' \to \\ C' \cup C'' \cup (\bigcup_{k_3} C_{k_3}) \cup (\bigcup_{k_3} C'_{k_3})\end{array}}$$

  Let us write the proof for the case where $\Gamma(k_1, k_2) \neq \text{seskey}^{\texttt{HH},\infty}(T)$. The other case is similar, although the keys are renamed, and slightly easier, since by well-formedness of Γ there are no k_3 satisfying the assumptions of the last two premises.

  We thus have $[\,k_1\,]^{\Gamma}_i = k_1$, $[\,k_2\,]^{\Gamma}_i = k_2$, and for any k_3 satisfying the assumptions of either of the last two premises, $[\,k_3\,]^{\Gamma}_i = k_3$.

  Since

$$[\,\Gamma, x : T\,]^n_i = [\,\Gamma\,]^n_i, x_i : [\,T\,]^n,$$

  by applying the induction hypothesis to Π', there exist $C'_1 \subseteq [\,C'\,]^n_i$ and a proof Π'_1 of

$$[\,\Gamma\,]^n_i, x_i : [\,T\,]^n \vdash [\,P'\,]^{\Gamma,x:T}_i \sim [\,Q'\,]^{\Gamma,x:T}_i \to C'_1.$$

Similarly, there exist $C''_1 \subseteq [\, C'' \,]^n_i$ and a proof Π''_1 of

$$[\, \Gamma \,]^n_i \vdash [\, P'' \,]^\Gamma_i \sim [\, Q'' \,]^\Gamma_i \to C''_1.$$

Similarly, for each $k_3 \neq k_2$ such that $\Gamma(k_1, k_3) = \mathrm{key}^{\mathtt{HH}}(T')$ for some T', there exist $C_{k_3,1} \subseteq [\, C_{k_3} \,]^n_i$ and a proof Π_1^{1,k_3} of

$$[\, \Gamma \,]^n_i, x_i : [\, T' \,]^n \vdash [\, P' \,]^\Gamma_i \sim [\, Q'' \,]^\Gamma_i \to C_{k_3,1}.$$

Similarly, for each $k_3 \neq k_1$ such that $\Gamma(k_3, k_2) = \mathrm{key}^{\mathtt{HH}}(T')$ for some T', there exist $C'_{k_3,1} \subseteq \left[\, C'_{k_3} \,\right]^n_i$ and a proof Π_1^{2,k_3} of

$$[\, \Gamma \,]^n_i, x_i : [\, T' \,]^n \vdash [\, P'' \,]^\Gamma_i \sim [\, Q' \,]^\Gamma_i \to C'_{k_3,1}.$$

Moreover, by definition, $[\, \Gamma \,]^n_i(y_i) = \mathtt{LL}$, and for all l, T, and all keys k, k' that are either k_1, k_2, or a k_3 such as in the premises of the rule, if $\Gamma(k, k') <: \mathrm{key}^l(T)$ then $[\, \Gamma \,]^n_i(k, k') <: \mathrm{key}^l([\, T \,]^n)$.

In addition,

$$[\, P \,]^\Gamma_i = [\, \mathtt{let}\ x = \mathtt{dec}(y, k_1)\ \mathtt{in}\ P'\ \mathtt{else}\ P'' \,]^\Gamma_i = \mathtt{let}\ x_i = \mathtt{dec}(y_i, k_1)\ \mathtt{in}\ [\, P' \,]^\Gamma_i\ \mathtt{else}\ [\, P'' \,]^\Gamma_i,$$

i.e.

$$[\, P \,]^\Gamma_i = \mathtt{let}\ x_i = \mathtt{dec}(y_i, k_1)\ \mathtt{in}\ [\, P' \,]^{\Gamma, x:T}_i\ \mathtt{else}\ [\, P'' \,]^\Gamma_i.$$

Similarly, $[\, Q \,]^\Gamma_i = \mathtt{let}\ x_i = \mathtt{dec}(y_i, k_2)\ \mathtt{in}\ [\, Q' \,]^{\Gamma, x:T}_i\ \mathtt{else}\ [\, Q'' \,]^\Gamma_i$.

Therefore, using Π'_1, Π''_1, the Π_1^{1,k_3} and Π_1^{2,k_3}, and rule PLetDec, we have $[\, \Gamma \,]^n_i \vdash [\, P \,]^\Gamma_i \sim [\, Q \,]^\Gamma_i \to C'_1 \cup C''_1 \cup (\bigcup_{k_3} C_{k_3,1}) \cup (\bigcup_{k_3} C'_{k_3,1}) \subseteq [\, C \,]^n_i$.

- PLetAdecSame, PLetAdecDiff: these cases are similar to the PLetDec case.
- PLetLR: then $P = \mathtt{let}\ x = d(y)\ \mathtt{in}\ P'\ \mathtt{else}\ P''$, $Q = \mathtt{let}\ x = d(y)\ \mathtt{in}\ Q'\ \mathtt{else}\ Q''$ for some variable $x \in \mathcal{X}$ and some processes P', Q', P'', Q'', and

$$\Pi = \frac{\Gamma(y) = [\![\tau_m^{l,a} \,;\, \tau_p^{l',a}]\!] \vee \Gamma(y) <: \mathrm{key}^l(T) \qquad \dfrac{\Pi'}{\Gamma \vdash P'' \sim Q'' \to C}}{\Gamma \vdash P \sim Q \to C}$$

for some m, p.

By applying the induction hypothesis to Π', there exists $C' \subseteq [\, C \,]^n_i$ and a proof Π'' of $[\, \Gamma \,]^n_i \vdash [\, P'' \,]^\Gamma_i \sim [\, Q'' \,]^\Gamma_i \to C'$.

We have $[\, P \,]^\Gamma_i = [\, \mathtt{let}\ x = d(y)\ \mathtt{in}\ P'\ \mathtt{else}\ P'' \,]^\Gamma_i = \mathtt{let}\ x_i = d(y_i)\ \mathtt{in}\ [\, P' \,]^\Gamma_i\ \mathtt{else}\ [\, P'' \,]^\Gamma_i$. Similarly, $[\, Q \,]^\Gamma_i = \mathtt{let}\ x_i = d(y_i)\ \mathtt{in}\ [\, Q' \,]^\Gamma_i\ \mathtt{else}\ [\, Q'' \,]^\Gamma_i$.

Let us first prove the case where $\Gamma(y) = [\![\tau_m^{l,a} \,;\, \tau_p^{l',a}]\!]$.

We distinguish two cases, depending on whether the types in the refinement $[\![\tau_m^{l,a} \,;\, \tau_p^{l',a}]\!]$ are finite nonce types or infinite nonce types, *i.e.* whether a is 1 or ∞.

  - If a is 1: Then by definition of $[\, \Gamma \,]^n_i$, we have $[\, \Gamma \,]^n_i(y_i) = [\![\tau_m^{l,1} \,;\, \tau_p^{l',1}]\!]$.
    Therefore, using Π'' and rule PLetLR, we have $[\, \Gamma \,]^n_i \vdash [\, P \,]^\Gamma_i \sim [\, Q \,]^\Gamma_i \to C' \subseteq [\, C \,]^n_i$.

– If a is ∞: Then by definition of $[\,\Gamma\,]_i^n$, we have $[\,\Gamma\,]_i^n(y_i) = \bigvee_{1\leq j\leq n}[\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]$.
Let $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$. By definition, there exists $j \in [\![1,n]\!]$, such that $\Gamma'(y_i) = [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]$.
Using Π'' and Lemma A.7, there exist $C'_{\Gamma'} \subseteq [\,C\,]_i^n$ and a derivation $\Pi''_{\Gamma'}$ of $\Gamma' \vdash [\,P''\,]_i^\Gamma \sim [\,Q''\,]_i^\Gamma \to C'_{\Gamma'}$. Therefore, using rule PLetLR, we have, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, $\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C'_{\Gamma'} \subseteq [\,C\,]_i^n$. Thus, by Lemma A.30, we have

$$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C'',$$

where $C'' \subseteq [\,C\,]_i^n$, which proves the claim in this case.

We now prove the case where $\Gamma(y) <: \text{key}^l(T)$. By Lemma A.1, we thus have

$$\Gamma(y) \in \{\text{seskey}^{l,\infty}(T), \text{seskey}^{l,1}(T), \text{eqkey}^l(T), \text{key}^l(T)\}.$$

In all cases we have $\Gamma(y_i) <: \text{key}^l([\,T\,]^n)$. Hence using Π'' and rule PLetLR, we have $[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C''$.

- PIfL: then $P = \texttt{if}\ M = M'\ \texttt{then}\ P'\ \texttt{else}\ P''$, $Q = \texttt{if}\ N = N'\ \texttt{then}\ Q'\ \texttt{else}\ Q''$ for some messages M, N, M', N', and some processes P', Q', P'', Q'', and

$$\Pi = \dfrac{\dfrac{\Pi''}{\Gamma \vdash P'' \sim Q'' \to C''} \quad \dfrac{\dfrac{\Pi'}{\Gamma \vdash P' \sim Q' \to C'}}{\Gamma \vdash M \sim N : \mathtt{LL} \to c} \quad \dfrac{\Pi_2}{\Gamma \vdash M' \sim N' : \mathtt{LL} \to c'}}{\Gamma \vdash P \sim Q \to C = (C' \cup C'') \cup_\forall (c \cup c')}.$$

By applying the induction hypothesis to Π', there exists $C''' \subseteq [\,C'\,]_i^n$ and a proof Π''' of $[\,\Gamma\,]_i^n \vdash [\,P'\,]_i^\Gamma \sim [\,Q'\,]_i^\Gamma \to C'''$. Similarly, there exists $C'''' \subseteq [\,C''\,]_i^n$ and a proof Π'''' of $[\,\Gamma\,]_i^n \vdash [\,P''\,]_i^\Gamma \sim [\,Q''\,]_i^\Gamma \to C''''$.
Hence, by Lemma A.7, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, there exist $C_{\Gamma'} \subseteq C'''$ and a proof $\Pi_{1,\Gamma'}$ of $\Gamma' \vdash [\,P'\,]_i^\Gamma \sim [\,Q'\,]_i^\Gamma \to C_{\Gamma'}$; as well as $C'_{\Gamma'} \subseteq C''''$ and a proof $\Pi_{2,\Gamma'}$ of $\Gamma' \vdash [\,P''\,]_i^\Gamma \sim [\,Q''\,]_i^\Gamma \to C'_{\Gamma'}$.
Moreover, by Lemma A.27, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, there exists a proof $\Pi'_{1,\Gamma'}$ of $\Gamma' \vdash [\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma : \mathtt{LL} \to [\,c\,]_i^\Gamma$. Similarly, there exists a proof $\Pi'_{2,\Gamma'}$ of $\Gamma' \vdash [\,M'\,]_i^\Gamma \sim [\,N'\,]_i^\Gamma : \mathtt{LL} \to [\,c'\,]_i^\Gamma$.
In addition,

$$[\,P\,]_i^\Gamma = [\,\texttt{if}\ M = M'\ \texttt{then}\ P'\ \texttt{else}\ P''\,]_i^\Gamma = \texttt{if}\ [\,M\,]_i^\Gamma = [\,M'\,]_i^\Gamma\ \texttt{then}\ [\,P'\,]_i^\Gamma\ \texttt{else}\ [\,P''\,]_i^\Gamma.$$

Similarly, $[\,Q\,]_i^\Gamma = \texttt{if}\ [\,N\,]_i^\Gamma = [\,N'\,]_i^\Gamma\ \texttt{then}\ [\,Q'\,]_i^\Gamma\ \texttt{else}\ [\,Q''\,]_i^\Gamma$.
Therefore, using $\Pi_{1,\Gamma'}$, $\Pi_{2,\Gamma'}$, $\Pi'_{1,\Gamma'}$, $\Pi'_{2,\Gamma'}$ and rule PIfL, we have

$$\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to (C_{\Gamma'} \cup C'_{\Gamma'}) \cup_\forall ([\,c\,]_i^\Gamma \cup [\,c'\,]_i^\Gamma) \subseteq (C''' \cup C'''') \cup_\forall ([\,c\,]_i^\Gamma \cup [\,c'\,]_i^\Gamma).$$

Thus by Lemma A.30, there exists $C_1 \subseteq (C''' \cup C'''') \cup_\forall ([\,c\,]_i^\Gamma \cup [\,c'\,]_i^\Gamma)$ such that

$$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_1.$$

Finally, $[\,C\,]_i^n = [\,(C' \cup C'') \cup_\forall (c \cup c')\,]_i^n = ([\,C'\,]_i^n \cup [\,C''\,]_i^n) \cup_\forall ([\,c\,]_i^\Gamma \cup [\,c'\,]_i^\Gamma)$ (by Lemma A.31, whose conditions are satisfied, by Lemma A.12). Hence $(C''' \cup C'''') \cup_\forall ([\,c\,]_i^\Gamma \cup [\,c'\,]_i^\Gamma) \subseteq [\,C\,]_i^n$, which proves the claim.

- PIFP: then $P = \texttt{if}\ M = t\ \texttt{then}\ P'\ \texttt{else}\ P''$, $Q = \texttt{if}\ N = t\ \texttt{then}\ Q'\ \texttt{else}\ Q''$ for some messages M, N, some $t \in \mathcal{K} \cup \mathcal{N} \cup \mathcal{C}$, and some processes P', Q', P'', Q'', and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash P' \sim Q' \to C'} \quad \dfrac{\Pi''}{\Gamma \vdash P'' \sim Q'' \to C''} \quad \dfrac{\Pi_1}{\Gamma \vdash M \sim N : \mathtt{LL} \to c} \quad \dfrac{\Pi_2}{\Gamma \vdash t \sim t : \mathtt{LL} \to c'}}{\Gamma \vdash P \sim Q \to C = C' \cup C''}.$$

  By applying the induction hypothesis to Π', there exist $C''' \subseteq [\,C'\,]_i^n$ and a proof Π''' of the judgement $[\,\Gamma\,]_i^n \vdash [\,P'\,]_i^\Gamma \sim [\,Q'\,]_i^\Gamma \to C'''$. Similarly, there exist $C'''' \subseteq [\,C''\,]_i^n$ and a proof Π'''' of $[\,\Gamma\,]_i^n \vdash [\,P''\,]_i^\Gamma \sim [\,Q''\,]_i^\Gamma \to C''''$.

  Hence, by Lemma A.7, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, there exist $C_{\Gamma'} \subseteq C'''$ and a proof $\Pi_{1,\Gamma'}$ of the judgement $\Gamma' \vdash [\,P'\,]_i^\Gamma \sim [\,Q'\,]_i^\Gamma \to C_{\Gamma'}$; as well as $C'_{\Gamma'} \subseteq C''''$ and a proof $\Pi_{2,\Gamma'}$ of $\Gamma' \vdash [\,P''\,]_i^\Gamma \sim [\,Q''\,]_i^\Gamma \to C'_{\Gamma'}$.

  Moreover, by Lemma A.27, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, there exists a proof $\Pi'_{1,\Gamma'}$ of the judgement $\Gamma' \vdash [\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma : \mathtt{LL} \to [\,c\,]_i^\Gamma$. Similarly, there exists a proof $\Pi'_{2,\Gamma'}$ of $\Gamma' \vdash [\,t\,]_i^\Gamma \sim [\,t\,]_i^\Gamma : \mathtt{LL} \to [\,c'\,]_i^\Gamma$.

  Since $t \in \mathcal{K} \cup \mathcal{N} \cup \mathcal{C}$, we also have $[\,t\,]_i^\Gamma \in \mathcal{K} \cup \mathcal{N} \cup \mathcal{C}$.

  In addition,

$$[\,P\,]_i^\Gamma = [\,\texttt{if}\ M = M'\ \texttt{then}\ P'\ \texttt{else}\ P''\,]_i^\Gamma = \texttt{if}\ [\,M\,]_i^\Gamma = [\,M'\,]_i^\Gamma\ \texttt{then}\ [\,P'\,]_i^\Gamma\ \texttt{else}\ [\,P''\,]_i^\Gamma.$$

  Similarly, $[\,Q\,]_i^\Gamma = \texttt{if}\ [\,N\,]_i^\Gamma = [\,N'\,]_i^\Gamma\ \texttt{then}\ [\,Q'\,]_i^\Gamma\ \texttt{else}\ [\,Q''\,]_i^\Gamma$.

  Therefore, using $\Pi_{1,\Gamma'}$, $\Pi_{2,\Gamma'}$, $\Pi'_{1,\Gamma'}$, $\Pi'_{2,\Gamma'}$ and rule PIFP, we have for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$

$$\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_{\Gamma'} \cup C'_{\Gamma'} \subseteq C''' \cup C''''.$$

  Thus by Lemma A.30, there exists $C_1 \subseteq (C''' \cup C'''')$ such that

$$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_1.$$

  Finally, $[\,C\,]_i^n = [\,C' \cup C''\,]_i^n = [\,C'\,]_i^n \cup [\,C''\,]_i^n$ (by Lemma A.31). Hence $(C''' \cup C'''') \subseteq [\,C\,]_i^n$, which proves the claim.

- PIFLR: then $P = \texttt{if}\ M_1 = M_2\ \texttt{then}\ P_\top\ \texttt{else}\ P_\bot$, $Q = \texttt{if}\ N_1 = N_2\ \texttt{then}\ Q_\top\ \texttt{else}\ Q_\bot$ for some messages M_1, N_1, M_2, N_2, and some processes $P_\top$, $Q_\top$, $P_\bot$, $Q_\bot$, and there exist m, p, m', p' such that

$$\Pi = \frac{\dfrac{\Pi_1}{\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!] \to c} \quad \dfrac{\Pi_2}{\Gamma \vdash M_2 \sim N_2 : [\![\tau_{m'}^{l'',1}\,;\,\tau_{p'}^{l''',1}]\!] \to c'} \quad b = (\tau_m^{l,1} \stackrel{?}{=} \tau_{m'}^{l'',1}) \quad b' = (\tau_p^{l'',1} \stackrel{?}{=} \tau_{p'}^{l''',1}) \quad \dfrac{\Pi'}{\Gamma \vdash P_b \sim Q_{b'} \to C}}{\Gamma \vdash P \sim Q \to C}.$$

By applying the induction hypothesis to Π', there exist $C' \subseteq [\, C \,]_i^n$ and a proof Π'' of the judgement $[\, \Gamma \,]_i^n \vdash [\, P_b \,]_i^\Gamma \sim [\, Q_{b'} \,]_i^\Gamma \to C'$.

Hence, by Lemma A.7, for all $\Gamma' \in \text{branches}([\, \Gamma \,]_i^n)$, there exist $C_{\Gamma'} \subseteq C'$, and a proof $\Pi_{\Gamma'}$ of $\Gamma' \vdash [\, P_b \,]_i^\Gamma \sim [\, Q_{b'} \,]_i^\Gamma \to C_{\Gamma'}$.

Moreover, by Lemma A.27 applied to Π_1, for all $\Gamma' \in \text{branches}([\, \Gamma \,]_i^n)$, there exists a proof $\Pi'_{1,\Gamma'}$ of the judgement $\Gamma' \vdash [\, M_1 \,]_i^\Gamma \sim [\, N_1 \,]_i^\Gamma : [\![\tau_m^{l,1} ; \tau_p^{l',1}]\!] \to [\, c \,]_i^\Gamma$. Similarly, there exists a proof $\Pi'_{2,\Gamma'}$ of $\Gamma' \vdash [\, M_2 \,]_i^\Gamma \sim [\, N_2 \,]_i^\Gamma : [\![\tau_{m'}^{l'',1} ; \tau_{p'}^{l''',1}]\!] \to [\, c' \,]_i^\Gamma$.

In addition,

$$[\, P \,]_i^\Gamma = [\, \texttt{if } M_1 = M_2 \texttt{ then } P_\top \texttt{ else } P_\bot \,]_i^\Gamma = \texttt{if } [\, M_1 \,]_i^\Gamma = [\, M_2 \,]_i^\Gamma \texttt{ then } [\, P_\top \,]_i^\Gamma \texttt{ else } [\, P_\bot \,]_i^\Gamma.$$

Similarly, $[\, Q \,]_i^\Gamma = \texttt{if } [\, N_1 \,]_i^\Gamma = [\, N_2 \,]_i^\Gamma \texttt{ then } [\, Q_\top \,]_i^\Gamma \texttt{ else } [\, Q_\bot \,]_i^\Gamma$.

Therefore, using $\Pi_{\Gamma'}$, $\Pi'_{1,\Gamma'}$, $\Pi'_{2,\Gamma'}$ and rule PIFLR, we have for all $\Gamma' \in \text{branches}([\, \Gamma \,]_i^n)$

$$\Gamma' \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C_{\Gamma'} \subseteq C' \subseteq [\, C \,]_i^n.$$

Thus by Lemma A.30, there exists $C_1 \subseteq [\, C \,]_i^n$ such that

$$[\, \Gamma \,]_i^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C_1,$$

which proves the claim.

- PIFS: then $P = \texttt{if } M = M' \texttt{ then } P' \texttt{ else } P''$, $Q = \texttt{if } N = N' \texttt{ then } Q' \texttt{ else } Q''$ for some messages M, N, M', N', and some processes P', Q', P'', Q'', and

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash P'' \sim Q'' \to C} \qquad \dfrac{\Pi_1}{\Gamma \vdash M \sim N : \texttt{LL} \to c} \qquad \dfrac{\Pi_2}{\Gamma \vdash M' \sim N' : \texttt{HH} \to c'}}{\Gamma \vdash P \sim Q \to C}.$$

By applying the induction hypothesis to Π', there exists $C' \subseteq [\, C \,]_i^n$ and a proof Π'' of the judgement $[\, \Gamma \,]_i^n \vdash [\, P'' \,]_i^\Gamma \sim [\, Q'' \,]_i^\Gamma \to C'$.

Hence, by Lemma A.7, for all $\Gamma' \in \text{branches}([\, \Gamma \,]_i^n)$, there exist $C_{\Gamma'} \subseteq C'$ and a proof $\Pi_{\Gamma'}$ of the judgement $\Gamma' \vdash [\, P' \,]_i^\Gamma \sim [\, Q' \,]_i^\Gamma \to C_{\Gamma'}$.

Moreover, by Lemma A.27, for all $\Gamma' \in \text{branches}([\, \Gamma \,]_i^n)$, there exists a proof $\Pi'_{1,\Gamma'}$ of the judgement $\Gamma' \vdash [\, M \,]_i^\Gamma \sim [\, N \,]_i^\Gamma : \texttt{LL} \to [\, c \,]_i^\Gamma$. Similarly, there exists a proof $\Pi'_{2,\Gamma'}$ of $\Gamma' \vdash [\, M' \,]_i^\Gamma \sim [\, N' \,]_i^\Gamma : \texttt{HH} \to [\, c' \,]_i^\Gamma$.

In addition,

$$[\, P \,]_i^\Gamma = [\, \texttt{if } M = M' \texttt{ then } P' \texttt{ else } P'' \,]_i^\Gamma = \texttt{if } [\, M \,]_i^\Gamma = [\, M' \,]_i^\Gamma \texttt{ then } [\, P' \,]_i^\Gamma \texttt{ else } [\, P'' \,]_i^\Gamma.$$

Similarly, $[\, Q \,]_i^\Gamma = \texttt{if } [\, N \,]_i^\Gamma = [\, N' \,]_i^\Gamma \texttt{ then } [\, Q' \,]_i^\Gamma \texttt{ else } [\, Q'' \,]_i^\Gamma$.

Therefore, using $\Pi_{\Gamma'}$, $\Pi'_{1,\Gamma'}$, $\Pi'_{2,\Gamma'}$ and rule PIFS, we have for all $\Gamma' \in \text{branches}([\, \Gamma \,]_i^n)$

$$\Gamma' \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C_{\Gamma'} \subseteq C' \subseteq [\, C \,]_i^n.$$

Thus by Lemma A.30, there exists $C_1 \subseteq [\, C \,]_i^n$ such that

$$[\, \Gamma \,]_i^n \vdash [\, P \,]_i^\Gamma \sim [\, Q \,]_i^\Gamma \to C_1,$$

which proves the claim.

- PIFI: then $P = \texttt{if}\ M = M'\ \texttt{then}\ P'\ \texttt{else}\ P''$, $Q = \texttt{if}\ N = N'\ \texttt{then}\ Q'\ \texttt{else}\ Q''$ for some messages M, N, M', N', and some processes P', Q', P'', Q'', and there exist types T, T', and names m, p, such that

$$\Pi = \frac{\dfrac{\Pi'}{\Gamma \vdash P'' \sim Q'' \to C} \qquad \dfrac{\Pi_1}{\Gamma \vdash M \sim N : T * T' \to c} \qquad \dfrac{\Pi_2}{\Gamma \vdash M' \sim N' : [\![\tau_m^{l,a}\,;\,\tau_p^{l',a}]\!] \to c'}}{\Gamma \vdash P \sim Q \to C}.$$

  By applying the induction hypothesis to Π', there exist $C' \subseteq [\,C\,]_i^n$ and a proof Π'' of the judgement $[\,\Gamma\,]_i^n \vdash [\,P''\,]_i^\Gamma \sim [\,Q''\,]_i^\Gamma \to C'$.

  Let $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$. By applying Lemma A.7 to Π'', there exists $C_{\Gamma'} \subseteq C'$, such that there exists a proof $\Pi_{\Gamma'}$ of the judgement $\Gamma' \vdash [\,P''\,]_i^\Gamma \sim [\,Q''\,]_i^\Gamma \to C_{\Gamma'}$.

  Moreover, by Lemma A.27 applied to the proof Π_1, there exists a proof $\Pi'_{1,\Gamma'}$ of the type judgement $\Gamma' \vdash [\,M\,]_i^\Gamma \sim [\,N\,]_i^\Gamma : [\,T\,]^n * [\,T'\,]^n \to [\,c\,]_i^\Gamma$. Similarly, there exists a proof $\Pi'_{2,\Gamma'}$ of the judgement $\Gamma' \vdash [\,M'\,]_i^\Gamma \sim [\,N'\,]_i^\Gamma : \left[\,[\![\tau_m^{l,a}\,;\,\tau_p^{l',a}]\!]\,\right]^n) \to [\,c'\,]_i^\Gamma$.

  In addition,

$$[\,P\,]_i^\Gamma = [\,\texttt{if}\ M = M'\ \texttt{then}\ P'\ \texttt{else}\ P''\,]_i^\Gamma = \texttt{if}\ [\,M\,]_i^\Gamma = [\,M'\,]_i^\Gamma\ \texttt{then}\ [\,P'\,]_i^\Gamma\ \texttt{else}\ [\,P''\,]_i^\Gamma.$$

  Similarly, $[\,Q\,]_i^\Gamma = \texttt{if}\ [\,N\,]_i^\Gamma = [\,N'\,]_i^\Gamma\ \texttt{then}\ [\,Q'\,]_i^\Gamma\ \texttt{else}\ [\,Q''\,]_i^\Gamma$.

  We distinguish two cases.

  - If a is 1: Then $\left[\,[\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!]\,\right]^n = [\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!]$, and using $\Pi_{\Gamma'}$, $\Pi'_{1,\Gamma'}$, $\Pi'_{2,\Gamma'}$ and rule PIFI, we have for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$

$$\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_{\Gamma'} \subseteq C' \subseteq [\,C\,]_i^n.$$

    Thus by Lemma A.30, there exists $C_1 \subseteq [\,C\,]_i^n$ such that

$$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_1,$$

    which proves the claim in this case.
  - If a is ∞: Moreover, by applying Lemma A.5 to $\Pi'_{2,\Gamma'}$, there exists a type

$$T'' \in \text{branches}(\left[\,[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]\,\right]^n),$$

    such that there exists a proof $\Pi''_{2,\Gamma'}$ of $\Gamma' \vdash [\,M'\,]_i^\Gamma \sim [\,N'\,]_i^\Gamma : T'' \to [\,c'\,]_i^\Gamma$.

    By definition, $\left[\,[\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]\,\right]^n = \bigvee_{1\le j\le n}[\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]$. Therefore, by definition of branches, there exists j such that $T'' = [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]$.

    Hence, using $\Pi_{\Gamma'}$, $\Pi'_{1,\Gamma'}$, $\Pi''_{2,\Gamma'}$, by rule PIFI, we have for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$ that

$$\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_{\Gamma'} \subseteq C' \subseteq [\,C\,]_i^n.$$

    Thus by Lemma A.30, there exists $C_1 \subseteq [\,C\,]_i^n$ such that

$$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_1$$

    which proves the claim in this case.

- PIFLR*: then $P = \texttt{if } M_1 = M_2 \texttt{ then } P_\top \texttt{ else } P_\bot$, $Q = \texttt{if } N_1 = N_2 \texttt{ then } Q_\top \texttt{ else } Q_\bot$ for some messages M_1, N_1, M_2, N_2, and some processes $P_\top$, $Q_\top$, $P_\bot$, $Q_\bot$, and there exist m, p, l, l' such that

$$\Pi = \dfrac{\dfrac{\Pi_2}{\Gamma \vdash M_2 \sim N_2 : [\![\tau_m^{l,\infty}\,;\,\tau_{p'}^{l',\infty}]\!] \to \emptyset} \quad \dfrac{\Pi_\top}{\Gamma \vdash P_\top \sim Q_\top \to C_1} \quad \dfrac{\dfrac{\Pi_1}{\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!] \to \emptyset}}{\Gamma \vdash P_\bot \sim Q_\bot \to C_2}}{\Gamma \vdash P \sim Q \to C = C_1 \cup C_2}.$$

  By applying the induction hypothesis to $\Pi_\top$, there exist $C' \subseteq [\,C_1\,]_i^n \subseteq [\,C\,]_i^n$ and a proof Π' of $[\,\Gamma\,]_i^n \vdash [\,P_\top\,]_i^\Gamma \sim [\,Q_\top\,]_i^\Gamma \to C'$. Similarly with $\Pi_\bot$, there exist $C'' \subseteq [\,C_2\,]_i^n \subseteq [\,C\,]_i^n$ and a proof Π'' of $[\,\Gamma\,]_i^n \vdash [\,P_\bot\,]_i^\Gamma \sim [\,Q_\bot\,]_i^\Gamma \to C''$.

  Hence, by Lemma A.7, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, there exist constraint sets $C_{1,\Gamma'} \subseteq C' (\subseteq [\,C\,]_i^n)$, and $C_{2,\Gamma'} \subseteq C'' (\subseteq [\,C\,]_i^n)$, and proofs $\Pi_{\top,\Gamma'}$ and $\Pi_{\bot,\Gamma'}$ of $\Gamma' \vdash [\,P_\top\,]_i^\Gamma \sim [\,Q_\top\,]_i^\Gamma \to C_{1,\Gamma'}$ and $\Gamma' \vdash [\,P_\bot\,]_i^\Gamma \sim [\,Q_\bot\,]_i^\Gamma \to C_{2,\Gamma'}$.

  Moreover, by Lemma A.27 applied to Π_1, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, there exists a proof $\Pi'_{1,\Gamma'}$ of $\Gamma' \vdash [\,M_1\,]_i^\Gamma \sim [\,N_1\,]_i^\Gamma : \bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!] \to [\,c\,]_i^\Gamma$. Similarly, there exists a proof $\Pi'_{2,\Gamma'}$ of $\Gamma' \vdash [\,M_2\,]_i^\Gamma \sim [\,N_2\,]_i^\Gamma : \bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!] \to [\,c'\,]_i^\Gamma$.

  Let $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$. By Lemma A.5, there exists $T \in \text{branches}(\bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!])$ such that there exists a proof $\Pi''_{1,\Gamma'}$ of $\Gamma' \vdash [\,M_1\,]_i^\Gamma \sim [\,N_1\,]_i^\Gamma : T \to [\,c\,]_i^\Gamma$.

  Similarly, there exists $T' \in \text{branches}(\bigvee_{1 \le j \le n} [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!])$ such that there exists a proof $\Pi''_{2,\Gamma'}$ of $\Gamma' \vdash [\,M_2\,]_i^\Gamma \sim [\,N_2\,]_i^\Gamma : T' \to [\,c\,]_i^\Gamma$.

  By definition of branches, there exist j, j' such that $T = [\![\tau_{m_j}^{l,1}\,;\,\tau_{p_j}^{l',1}]\!]$ and $T' = [\![\tau_{m_{j'}}^{l,1}\,;\,\tau_{p_{j'}}^{l',1}]\!]$.

  In addition,

  $$[\,P\,]_i^\Gamma = [\,\texttt{if } M_1 = M_2 \texttt{ then } P_\top \texttt{ else } P_\bot\,]_i^\Gamma = \texttt{if } [\,M_1\,]_i^\Gamma = [\,M_2\,]_i^\Gamma \texttt{ then } [\,P_\top\,]_i^\Gamma \texttt{ else } [\,P_\bot\,]_i^\Gamma.$$

  Similarly, $[\,Q\,]_i^\Gamma = \texttt{if } [\,N_1\,]_i^\Gamma = [\,N_2\,]_i^\Gamma \texttt{ then } [\,Q_\top\,]_i^\Gamma \texttt{ else } [\,Q_\bot\,]_i^\Gamma$.

  Therefore, using $\Pi_{\top,\Gamma'}$, $\Pi_{\bot,\Gamma'}$, $\Pi''_{1,\Gamma'}$, $\Pi''_{2,\Gamma'}$ and rule PIFLR, either $j = j'$ and we have

  $$\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_{1,\Gamma'} (\subseteq [\,C\,]_i^n)$$

  or $j \neq j'$ and we have

  $$\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_{2,\Gamma'} (\subseteq [\,C\,]_i^n).$$

  This holds for any $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$.

  Thus by Lemma A.30, there exists $C' \subseteq [\,C\,]_i^n$ such that

  $$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C'$$

  which proves the claim in this case.

- PIFLR'*: then $P = \texttt{if } M_1 = M_2 \texttt{ then } P_\top \texttt{ else } P_\bot$, $Q = \texttt{if } N_1 = N_2 \texttt{ then } Q_\top \texttt{ else } Q_\bot$ for some messages M_1, N_1, M_2, N_2, and some processes $P_\top$, $Q_\top$, $P_\bot$, $Q_\bot$, and there exist names m, p, m', p' such that

$$\Pi = \dfrac{\dfrac{\dfrac{\Pi_1}{\Gamma \vdash M_1 \sim N_1 : [\![\tau_m^{l,a}\,;\,\tau_p^{l',a}]\!] \to \emptyset}\qquad \dfrac{\Pi_2}{\Gamma \vdash M_2 \sim N_2 : [\![\tau_{m'}^{l'',a'}\,;\,\tau_{p'}^{l''',a'}]\!] \to \emptyset}}{\Gamma \vdash P \sim Q \to C}\qquad \dfrac{\Pi'}{\Gamma \vdash P_\perp \sim Q_\perp \to C}}{\Gamma \vdash P \sim Q \to C}.$$

By applying the induction hypothesis to Π', there exist $C' \subseteq [\,C\,]_i^n$ and a proof Π'' of the judgement $[\,\Gamma\,]_i^n \vdash [\,P_\perp\,]_i^\Gamma \sim [\,Q_\perp\,]_i^\Gamma \to C'$.

Hence, by Lemma A.7, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, there exist $C_{\Gamma'} \subseteq C' (\subseteq [\,C\,]_i^n)$, and a proof $\Pi_{\Gamma'}$ of $\Gamma' \vdash [\,P_\perp\,]_i^\Gamma \sim [\,Q_\perp\,]_i^\Gamma \to C_{\Gamma'}$.

Moreover, by Lemma A.27 applied to Π_1, for all $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$, there exists a proof $\Pi'_{1,\Gamma'}$ of $\Gamma' \vdash [\,M_1\,]_i^\Gamma \sim [\,N_1\,]_i^\Gamma : \left[\!\left[[\![\tau_m^{l,a}\,;\,\tau_p^{l',a}]\!] \right]\!\right]^n \to [\,c\,]_i^\Gamma$. Similarly, there exists a proof $\Pi'_{2,\Gamma'}$ of $\Gamma' \vdash [\,M_2\,]_i^\Gamma \sim [\,N_2\,]_i^\Gamma : \left[\!\left[[\![\tau_{m'}^{l'',a'}\,;\,\tau_{p'}^{l''',a'}]\!] \right]\!\right]^n \to [\,c'\,]_i^\Gamma$.

We distinguish several cases, depending on a and a'.

- if a and a' are both 1: Then this rule is a particular case of rule PIFLR, and the result is proved in a similar way.
- if a is 1 and a' is ∞: Then $\left[\!\left[[\![\tau_m^{l,a}\,;\,\tau_p^{l',a}]\!] \right]\!\right]^n = \left[\!\left[[\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!] \right]\!\right]^n$, and

  $$\left[\!\left[[\![\tau_{m'}^{l'',a}\,;\,\tau_{p'}^{l''',a}]\!] \right]\!\right]^n = \bigvee_{1\le j\le n} [\![\tau_{m'_j}^{l'',1}\,;\,\tau_{p_j}^{l''',1}]\!].$$

  Let $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$.
  By Lemma A.5, using $\Pi'_{2,\Gamma'}$, there exists $j \in [\![1,n]\!]$ such that there exists a proof $\Pi''_{2,\Gamma'}$ of $\Gamma' \vdash [\,M_2\,]_i^\Gamma \sim [\,N_2\,]_i^\Gamma : [\![\tau_{m'_j}^{l'',1}\,;\,\tau_{p'_j}^{l''',1}]\!] \to [\,c'\,]_i^\Gamma$.
  In addition,

  $$[\,P\,]_i^\Gamma = [\,\texttt{if } M_1 = M_2 \texttt{ then } P_\top \texttt{ else } P_\perp\,]_i^\Gamma = \texttt{if } [\,M_1\,]_i^\Gamma = [\,M_2\,]_i^\Gamma \texttt{ then } [\,P_\top\,]_i^\Gamma \texttt{ else } [\,P_\perp\,]_i^\Gamma.$$

  Similarly, $[\,Q\,]_i^\Gamma = \texttt{if } [\,N_1\,]_i^\Gamma = [\,N_2\,]_i^\Gamma \texttt{ then } [\,Q_\top\,]_i^\Gamma \texttt{ else } [\,Q_\perp\,]_i^\Gamma$.
  For any $j \in [\![1,n]\!]$, $\tau_m^{l,1} \neq \tau_{m'_j}^{l'',1}$; and $\tau_p^{l',1} \neq \tau_{p'_j}^{l''',1}$.
  Therefore, using $\Pi_{\Gamma'}$, $\Pi'_{1,\Gamma'}$, $\Pi''_{2,\Gamma'}$ and rule PIFLR, we have

  $$\Gamma' \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C_{\Gamma'} (\subseteq [\,C\,]_i^n).$$

  This holds for any $\Gamma' \in \text{branches}([\,\Gamma\,]_i^n)$.
  Thus by Lemma A.30, there exists $C' \subseteq [\,C\,]_i^n$ such that

  $$[\,\Gamma\,]_i^n \vdash [\,P\,]_i^\Gamma \sim [\,Q\,]_i^\Gamma \to C'$$

  which proves the claim in this case.
- if a is ∞ and a' is 1: This case is similar to the symmetric one.
- if a and a' both are ∞: This case is similar to the case where a is 1 and a' is ∞.

- PIFALL: this case is similar to the PIFL case.

□

Theorem A.4 (Typing n sessions)**.** *For all* Γ, P, Q *and* C, *such that*

$$\Gamma \vdash P \sim Q \to C$$

then for all $n \geq 1$, *there exists* $C' \subseteq \cup_{\times 1 \leq i \leq n} [\, C \,]^n_i$ *such that*

$$[\, \Gamma \,]^n \vdash [\, P \,]^\Gamma_1 \mid \ldots \mid [\, P \,]^\Gamma_n \sim [\, Q \,]^\Gamma_1 \mid \ldots \mid [\, Q \,]^\Gamma_n \to C'$$

where $[\, \Gamma \,]^n$ *is defined as* $\bigcup_{1 \leq i \leq n} [\, \Gamma \,]^n_i$.

Proof. Let us assume Γ, P, Q and C are such that

$$\Gamma \vdash P \sim Q \to C.$$

The claim clearly holds (using Theorem A.3) if $n = 1$. Let then $n \geq 2$.

Note that the union $\bigcup_{1 \leq i \leq n} [\, \Gamma \,]^n_i$ is well-defined, as for $i \neq j$, $\mathrm{dom}([\, \Gamma \,]^n_i) \cap \mathrm{dom}([\, \Gamma \,]^n_j) \subseteq \mathcal{K} \cup \mathcal{N}$, and the types associated to keys and nonces are the same in each $[\, \Gamma \,]^n_i$.

The property follows from Theorem A.3. Indeed, this theorem guarantees that for all $i \in [\![1, n]\!]$, there exists $C_i \subseteq [\, C \,]^n_i$ such that

$$[\, \Gamma \,]^n_i \vdash [\, P \,]^\Gamma_i i \sim [\, Q \,]^\Gamma_i i \to C_i.$$

By construction, all variables in $\mathrm{dom}([\, \Gamma \,]^n_i)$ are indexed with i, and all keys and nonces are either unindexed or indexed with i, and as we mentioned earlier, for all i, j, $[\, \Gamma \,]^n_i$ and $[\, \Gamma \,]^n_j$ have the same values on their common domain.

Hence we have $[\, \Gamma \,]^n = [\, \Gamma \,]^n_i \uplus (\bigcup_{j \neq i} ([\, \Gamma \,]^n_j)|_{\mathrm{dom}([\, \Gamma \,]^n_j) \setminus \mathrm{dom}([\, \Gamma \,]^n_i)})$. Therefore, by Lemma A.10, we have for all $i \in [\![1, n]\!]$

$$[\, \Gamma \,]^n \vdash [\, P \,]^\Gamma_i i \sim [\, Q \,]^\Gamma_i i \to C'_i$$

where

$$C'_i = \{(c, \Gamma' \cup \Gamma'') | (c, \Gamma') \in C_i \ \wedge \ \Gamma'' \in \mathrm{branches}(\bigcup_{j \neq i} ([\, \Gamma \,]^n_j)|_{d_{i,j}})\}$$

and

$$d_{i,j} = \mathrm{dom}([\, \Gamma \,]^n_j) \setminus \mathrm{dom}([\, \Gamma \,]^n_i).$$

Thus, by applying rule PPAR $n - 1$ times, we have

$$[\, \Gamma \,]^n \vdash [\, P \,]^\Gamma_i 1 \mid \ldots \mid [\, P \,]^\Gamma_i n \sim [\, Q \,]^\Gamma_i 1 \mid \ldots [\, Q \,]^\Gamma_i n \to \cup_{\times 1 \leq i \leq n} C'_i.$$

It only remains to be proved that $\cup_{\times 1 \leq i \leq n} C'_i \subseteq \cup_{\times 1 \leq i \leq n} [\, C \,]^n_i$. Since for all $i \in [\![1, n]\!]$ we have $C_i \subseteq [\, C \,]^n_i$, by Lemma A.11 we know that $\cup_{\times 1 \leq i \leq n} C_i \subseteq \cup_{\times 1 \leq i \leq n} [\, C \,]^n_i$.

Hence it suffices to show that $\cup_{\times 1 \leq i \leq n} C'_i \subseteq \cup_{\times 1 \leq i \leq n} C_i$.

Let $(c, \Gamma') \in \cup_{\times 1 \leq i \leq n} C'_i$. By definition there exist $(c_1, \Gamma_1) \in C'_1, \ldots, (c_n, \Gamma_n) \in C'_n$ such that $c = \bigcup_{1 \leq i \leq n} c_i$, $\Gamma' = \bigcup_{1 \leq i \leq n} \Gamma_i$, and for all $i \neq j$, Γ_i and Γ_j are compatible.

For all i, $(c_i, \Gamma_i) \in C'_i$. Thus by definition of C'_i there exist Γ'_i and Γ''_i such that $(c_i, \Gamma'_i) \in C_i$, $\Gamma''_i \in \mathrm{branches}(\bigcup_{j \neq i} ([\, \Gamma \,]^n_j)|_{d_{i,j}})$, and $\Gamma_i = \Gamma'_i \cup \Gamma''_i$. Since for all $i \neq j$, Γ_i and Γ_j are compatible, we know that Γ'_i and Γ'_j also are, as well as Γ'_i and Γ''_j.

Hence, $\Gamma' = \bigcup_{1\leq i\leq n}\Gamma_i = \bigcup_{1\leq i\leq n}(\Gamma'_i \cup \Gamma''_i) = (\bigcup_{1\leq i\leq n}\Gamma'_i) \cup (\bigcup_{1\leq i\leq n}\Gamma''_i)$.
Moreover, $(\bigcup_{1\leq i\leq n}\Gamma''_i) = (\bigcup_{1\leq i\leq n}\Gamma'_i)$. Indeed, they have the same domain, *i.e.*

$$\bigcup_{i\neq j} d_{i,j} = \bigcup_{i\neq j} \mathrm{dom}([\,\Gamma\,]^n_j)\backslash\mathrm{dom}([\,\Gamma\,]^n_i) = \bigcup_i \mathrm{dom}([\,\Gamma\,]^n_i)$$

since $n \geq 2$, and are compatible since for all $i \neq j$, Γ'_i and Γ''_j are compatible.

Thus $\Gamma' = (\bigcup_{1\leq i\leq n}\Gamma'_i)$, and since the Γ'_i are all pairwise compatible, and for all i, $(c_i, \Gamma'_i) \in C_i$, we have $(c, \Gamma') \in \cup_{\times 1\leq i\leq n} C_i$.

This proves that $\cup_{\times 1\leq i\leq n} C'_i \subseteq \cup_{\times 1\leq i\leq n} C_i$, which concludes the proof. □

Theorem A.5. *Consider P, Q, P' ,Q', C, C', such that P, Q and P', Q' do not share any variable. Consider Γ, containing only keys and nonces with types of the form $\tau^{l,1}_n$.*

Assume that P and Q only bind nonces and keys with infinite nonce types, i.e. *using* $\mathtt{new}\ m : \tau^{l,\infty}_m$ *and* $\mathtt{new}\ k : \mathrm{seskey}^{l,\infty}(T)$ *for some label l and type T; while P' and Q' only bind nonces and keys with finite types,* i.e. *using* $\mathtt{new}\ m : \tau^{l,1}_m$ *and* $\mathtt{new}\ k : \mathrm{seskey}^{l,1}(T)$.

Let us abbreviate by $\mathtt{new}\ \overline{n}$ *the sequence of declarations of each nonce $m \in \mathrm{dom}(\Gamma)$ and session key k such that $\Gamma(k,k) = \mathrm{seskey}^{l,1}(T)$ for some l, T. If*

- $\Gamma \vdash P \sim Q \to C$,
- $\Gamma \vdash P' \sim Q' \to C'$,
- $C' \cup_\times (\cup_{\times 1\leq i\leq n} [\,C\,]^n_i)$ *is consistent for all $n \geq 1$,*

then $$\mathtt{new}\ \overline{n}.\ ((!P) \mid P') \approx_t \mathtt{new}\ \overline{n}.\ ((!Q) \mid Q').$$

Proof. Note that since Γ only contains keys and nonces with finite types, for all i, $[\,P\,]^\Gamma_i = [\,P\,]^\emptyset_i$ is just P where all variables and some names have been α-renamed, and similarly for Q. Since P', Q' only contain nonces with finite types, $[\,P'\,]^\Gamma_1$ and $[\,Q'\,]^\Gamma_1$ are P', Q' where all variables have been α-renamed.

By Theorem A.4, we know that for all i, n,

$$[\,\Gamma\,]^n \vdash [\,P\,]^\Gamma_1 \mid \ldots \mid [\,P\,]^\Gamma_n \sim [\,Q\,]^\Gamma_1 \mid \ldots \mid [\,Q\,]^\Gamma_n \to C''$$

where $[\,\Gamma\,]^n = \bigcup_{1\leq i\leq n} [\,\Gamma\,]^n_i$, and $C'' \subseteq \cup_{\times 1\leq i\leq n} [\,C\,]^n_i$.

By Theorem A.3, there also exists $C''' \subseteq [\,C'\,]^n_1$, such that

$$[\,\Gamma\,]^n_1 \vdash [\,P'\,]^\Gamma_1 \sim [\,Q'\,]^\Gamma_1 \to C'''.$$

Therefore, by Lemma A.10, we have

$$[\,\Gamma\,]^n \vdash [\,P'\,]^\Gamma_1 \sim [\,Q'\,]^\Gamma_1 \to C''''$$

where C'''' is C''' where all the environments have been extended with $\bigcup_{1\leq i\leq n} ([\,\Gamma\,]^n_i)_{\mathcal{N},\mathcal{K}}$ (note that this environment still only contains nonces and keys).

Therefore, by rules PPar and PNew,

$$\Gamma' \vdash \mathtt{new}\ \overline{n}.\ ([\,P\,]^\Gamma_1 \mid \ldots \mid [\,P\,]^\Gamma_n) \mid [\,P'\,]^\Gamma_1 \sim \mathtt{new}\ \overline{n}.\ ([\,Q\,]^\Gamma_1 \mid \ldots \mid [\,Q\,]^\Gamma_n) \mid [\,Q'\,]^\Gamma_1 \to C'' \cup_\times C''''$$

where Γ' is the restriction of $[\,\Gamma\,]^n$ to keys.

If $[\,C'\,]_1^n \cup_\times (\cup_{\times 1 \le i \le n} [\,C\,]_i^n)$ is consistent, similarly to the reasoning in the proof of Theorem A.4, $C'' \cup_\times C''''$ also is.

Then, by Theorem A.2,

$$\mathtt{new}\ \overline{n}.\ ([\,P\,]_1^\Gamma \mid \ldots \mid [\,P\,]_n^\Gamma) \mid [\,P'\,]_1^\Gamma \approx_t \mathtt{new}\ \overline{n}.\ ([\,Q\,]_1^\Gamma \mid \ldots \mid [\,Q\,]_n^\Gamma) \mid [\,Q'\,]_1^\Gamma$$

which implies (since $[\,P'\,]_1^\Gamma$ is just a renaming of the variables in P') that

$$\mathtt{new}\ \overline{n}.\ ([\,P\,]_1^\Gamma \mid \ldots \mid [\,P\,]_n^\Gamma) \mid P' \approx_t \mathtt{new}\ \overline{n}.\ ([\,Q\,]_1^\Gamma \mid \ldots \mid [\,Q\,]_n^\Gamma) \mid Q'$$

Since $[\,P\,]_i^\Gamma$ and $[\,Q\,]_i^\Gamma$ are just α-renamings of P, Q, this implies that for all n,

$$\mathtt{new}\ \overline{n}.\ (P_1 \mid \ldots \mid P_n) \mid P' \approx_t \mathtt{new}\ \overline{n}.\ (Q_1 \mid \ldots \mid Q_n) \mid Q'$$

where $P_1 = \cdots = P_n = P$, and $Q_1 = \cdots = Q_n = Q$. Therefore

$$\mathtt{new}\ \overline{n}.\ ((!P) \mid P') \approx_t \mathtt{new}\ \overline{n}.\ ((!Q) \mid Q').$$

□

A.3 Checking consistency

In this section, we prove the soundness of the procedure `check_const` in the non-replicated case. We recall that a table displaying the correspondence between the numbering of the Lemmas and Theorems in Chapter 2 and here is provided at the beginning of Appendix A.

Note that we only consider constraints obtained by typing, and therefore such that Γ is well-formed, and such that there exists $c_\phi \subseteq c$ such that $\Gamma \vdash \phi_\ell(c) \sim \phi_r(c) : \mathtt{LL} \to c_\phi$.

Indeed, it is clear by induction on the typing rules for terms that:

$$\forall \Gamma, M, N, T, c.\quad \Gamma \vdash M \sim N : T \to c \Longrightarrow (\forall u \sim v \in c.\quad \exists c' \subseteq c.\quad \Gamma \vdash u \sim v : \mathtt{LL} \to c').$$

From this result, and using Lemmas A.3 and A.12, it follows clearly by induction on the typing rules for processes that

$$\forall \Gamma, P, Q, C.\quad \Gamma \vdash P \sim Q \to C \Longrightarrow (\forall (c, \Gamma') \in C.\quad \forall u \sim v \in c.\quad \exists c' \subseteq c.\quad \Gamma' \vdash u \sim v : \mathtt{LL} \to c').$$

Let us now prove that the procedure is correct for constraints without infinite nonce types, *i.e.* constraint sets C such that

$$\forall (c, \Gamma) \in C.\ \forall l, l', m, n.\ \Gamma(x) \neq [\![\tau_m^{l,\infty}\,;\,\tau_n^{l',\infty}]\!].$$

We fix such a constraint set C (obtained by typing).

Let $(c, \Gamma) \in C$. Let $(\overline{c}, \overline{\Gamma}) = \mathtt{step1}_\Gamma(c)$. Let us assume that $\mathtt{step2}_{\overline{\Gamma}}(\overline{c}) = \mathtt{true}$ and $\mathtt{step3}_{\overline{\Gamma}}(\overline{c}) = \mathtt{true}$.

Lemma A.32. *If $\overline{c}$ is consistent in $\overline{\Gamma}$, then c is consistent in Γ.*

Proof. Let c' be a set of constraints and Γ' be a typing environment such that $c' \subseteq c$, $\Gamma' \subseteq \Gamma$, ${\Gamma'}_{\mathcal{N},\mathcal{K}} = \Gamma_{\mathcal{N},\mathcal{K}}$ and $\mathrm{vars}(c') \subseteq \mathrm{dom}(\Gamma')$. Let σ, σ' be two substitutions such that ${\Gamma'}_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : {\Gamma'}_{\mathcal{X}} \to c_\sigma$ (for some set of constraints $c_\sigma \subseteq [\![c']\!]_{\sigma,\sigma'}$).

To prove the claim, we need to show that the frames $\phi^{\Gamma}_{\mathrm{LL}} \cup \phi_\ell([\![c']\!]_{\sigma,\sigma'}))$ and $(\phi^{\Gamma}_{\mathrm{LL}} \cup \phi_r([\![c']\!]_{\sigma,\sigma'}))$ are statically equivalent. Let D denote $\mathrm{dom}({\Gamma'}_{\mathcal{X}})(= \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma'))$.

For all $x \in F \cap D$, by definition of F, there exist m, n, l, l' such that $\Gamma(x) = [\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!]$. Thus, by well-typedness of σ, σ', there exists c_x such that $\Gamma \vdash \sigma(x) \sim \sigma'(x) : [\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!] \to c_x$. Hence, by Lemma A.14, since σ, σ' are ground, we have $\sigma(x) = m$ and $\sigma'(x) = n$. Therefore, $\sigma|_{D\cap F} = \sigma_F|_D$ and $\sigma'|_{D\cap F} = \sigma'_F|_D$.

Let c'' be the set $[\![c']\!]_{\sigma|_{D\cap F},\sigma'|_{D\cap F}}$. By Lemma A.11, we have $[\![c']\!]_{\sigma,\sigma'} = [\![c'']\!]_{\sigma|_{D\setminus F},\sigma'|_{D\setminus F}}$. We also have $c'' = [\![c']\!]_{\sigma_F|_D,\sigma'_F|_D}$, which is equal to $[\![c']\!]_{\sigma_F,\sigma'_F}$ since $\mathrm{vars}(c') \subseteq D$. Hence $c'' \subseteq \overline{c}$.

Let $\Gamma'' = \Gamma'|_{\mathrm{dom}(\Gamma')\setminus F}$. We have $\Gamma'' \subseteq \overline{\Gamma}$.

Moreover, since ${\Gamma'}_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : {\Gamma'}_{\mathcal{X}} \to c_\sigma$, it is clear from the definition of well-typedness for substitutions that we also have ${\Gamma''}_{\mathcal{N},\mathcal{K}} \vdash \sigma|_{D\setminus F} \sim \sigma'|_{D\setminus F} : {\Gamma''}_{\mathcal{X}} \to c'_\sigma$ for some $c'_\sigma \subseteq c_\sigma$. Note that $c'_\sigma \subseteq c_\sigma \subseteq [\![c']\!]_{\sigma,\sigma'} = [\![c'']\!]_{\sigma|_{D\setminus F},\sigma'|_{D\setminus F}}$. Finally, $\mathrm{vars}(c'') \subseteq \mathrm{vars}(c')\setminus F$ by definition of instantiation, thus $\mathrm{vars}(c'') \subseteq \mathrm{dom}(\Gamma'')$.

We have established that $c'' \subseteq \overline{c}$, $\Gamma'' \subseteq \overline{\Gamma}$, $\mathrm{vars}(c'') \subseteq \mathrm{dom}(\Gamma'')$, ${\Gamma''}_{\mathcal{N},\mathcal{K}} \vdash \sigma|_{D\setminus F} \sim \sigma'|_{D\setminus F} : {\Gamma''}_{\mathcal{X}} \to c'_\sigma$, and $c'_\sigma \subseteq [\![c'']\!]_{\sigma|_{D\setminus F},\sigma'|_{D\setminus F}}$. Therefore, by definition of the consistency of $\overline{c}$ in $\overline{\Gamma}$, the frames $\phi^{\overline{\Gamma}}_{\mathrm{LL}} \cup \phi_\ell([\![c'']\!]_{\sigma|_{D\setminus F},\sigma'|_{D\setminus F}})$ and $\phi^{\overline{\Gamma}}_{\mathrm{LL}} \cup \phi_r([\![c'']\!]_{\sigma|_{D\setminus F},\sigma'|_{D\setminus F}})$ are statically equivalent.

Since $\phi^{\overline{\Gamma}}_{\mathrm{LL}} \subseteq \phi^{\Gamma}_{\mathrm{LL}}$, that is to say that $\phi^{\Gamma}_{\mathrm{LL}} \cup \phi_\ell([\![c']\!]_{\sigma,\sigma'})$ and $\phi^{\Gamma}_{\mathrm{LL}} \cup \phi_r([\![c']\!]_{\sigma,\sigma'})$ are statically equivalent. This proves the consistency of c in Γ. □

Lemma A.33. *For all ground σ, σ' such that $\exists \Gamma' \subseteq \overline{\Gamma}.\ \exists c_\sigma.\ {\Gamma'}_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : {\Gamma'}_{\mathcal{X}} \to c_\sigma$, we have*

$$\mathtt{step2}_{\overline{\Gamma}}([\![\overline{c}]\!]_{\sigma,\sigma'}) = \mathtt{true}.$$

Proof. Let $M \sim N \in [\![\overline{c}]\!]_{\sigma,\sigma'}$. By definition there exist $M' \sim N' \in \overline{c}$ such that $M = M'\sigma$ and $N = N'\sigma'$. Since $\mathtt{step2}_{\overline{\Gamma}}(\overline{c}) = \mathtt{true}$, there are four cases for M' and N': they can be symmetric or asymmetric encryptions, hashes, or signatures. These four cases are similar, we write the proof for the asymmetric encryption case.

In this case, there exist M'_1, M'_2, N'_1, N'_2 such that $M' = \mathtt{aenc}(M'_1, M'_2)$ and $N' = \mathtt{aenc}(N'_1, N'_2)$. Let us show that $M'\sigma \sim N'\sigma'$ satisfies the conditions for $\mathtt{step2}_{\overline{\Gamma}}([\![\overline{c}]\!]_{\sigma,\sigma'})$.

Since $\mathtt{step2}_{\overline{\Gamma}}(\overline{c}) = \mathtt{true}$ we know that M'_1 and N'_1 contain directly under pairs

- a nonce n such that $\Gamma(n) = \tau_n^{\mathtt{HH},a}$
- or a secret key k such that $\exists T, k'.\Gamma(k,k') <: \mathrm{key}^{\mathtt{HH}}(T)$ or $\Gamma(k',k) <: \mathrm{key}^{\mathtt{HH}}(T)$,
- or a variable x such that $\exists m, n, a.\Gamma(x) = [\![\tau_m^{\mathtt{HH},a}\,;\,\tau_n^{\mathtt{HH},a}]\!]$,
- or a variable x such that $\exists T.\Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$.

In the fist two cases, $M'_1\sigma$ (resp. $N'_1\sigma'$) clearly contains the same nonce or keys under pairs. The last two cases are similar, we write the proof for the case of a variable x such that $\overline{\Gamma}(x) = [\![\tau_m^{\mathtt{HH},a}\,;\,\tau_n^{\mathtt{HH},a}]\!]$. Either $x \notin \mathrm{dom}(\sigma)$, and thus x still appears under pairs in $M'_1\sigma$; or $x \in \mathrm{dom}(\sigma)$. In that case, $\sigma(x)$ appear directly under pairs in $M'_1\sigma$ (resp. $\sigma'(x)$ in $N'_1\sigma'$). By assumption, there exists c' such that $\overline{\Gamma} \vdash \sigma(x) \sim \sigma'(x) : [\![\tau_m^{\mathtt{HH},a}\,;\,\tau_n^{\mathtt{HH},a}]\!] \to c'$. Hence, by Lemma A.14, $\sigma(x) = m$ (resp. $\sigma'(x) = n$) and $\overline{\Gamma}(n) = \tau_n^{\mathtt{HH},a}$ (resp. $\overline{\Gamma}(m) = \tau_m^{\mathtt{HH},a}$). Therefore, in all cases, $M'_1\sigma$ and $N'_1\sigma'$ satisfy the required conditions.

In addition, since $\mathtt{step2}_{\overline{\Gamma}}(\overline{c}) = \mathtt{true}$ we also know that M'_2 and N'_2 are either

- public keys $\mathtt{pk}(k)$, $\mathtt{pk}(k')$ where $\exists T.\Gamma(k,k') <: \mathrm{key}^{\mathtt{HH}}(T)$;
- or public keys $\mathtt{pk}(x)$, $\mathtt{pk}(x)$ where $\exists T.\Gamma(x) <: \mathrm{key}^{\mathtt{HH}}(T)$;
- or a variable x such that $\exists T, T'.\Gamma(x) = \mathrm{pkey}(T)$ and $T <: \mathrm{key}^{\mathtt{HH}}(T')$;

In the first case $M_2'\sigma = M_2'$ and $N_2'\sigma' = N_2'$ are still the same public keys. In the two other cases, similarly to the proof for M_1' and N_1', either $x \notin \mathrm{dom}(\sigma)$, and we also have $M_2'\sigma = M_2'$ and $N_2'\sigma' = N_2'$; or $x \in \mathrm{dom}(\sigma)$, and by well-typedness of σ, σ' and Lemma A.18, $M_2'\sigma$ and $N_2'\sigma'$ are keys of the right type. Therefore, in all cases, $M_2'\sigma$ and $N_2'\sigma'$ satisfy the required conditions.

Thus, $M'\sigma \sim N'\sigma'$ satisfies the conditions for $\mathtt{step2}_{\overline{\Gamma}}([\![\overline{c}]\!]_{\sigma,\sigma'})$. □

Lemma A.34. *There exists $c_\phi \subseteq \overline{c}$ such that $\overline{\Gamma} \vdash \phi^{\Gamma}_{\mathtt{LL}} \cup \phi_\ell(\overline{c}) \sim \phi^{\Gamma}_{\mathtt{LL}} \cup \phi_r(\overline{c}) : \mathtt{LL} \to c_\phi$.*

Proof. As explained previously, there exists $c'_\phi \subseteq c$ such that $\Gamma \vdash \phi_\ell(c) \sim \phi_r(c) : \mathtt{LL} \to c'_\phi$.

Moreover, we have by definition $\Gamma = \Gamma_F \uplus \overline{\Gamma}'$, where F is defined as in `step1` and Γ_F is the restriction of Γ to F, and for some $\overline{\Gamma}' \subseteq \overline{\Gamma}$. In addition $(\Gamma_F)_{\mathcal{N},\mathcal{K}} \vdash \sigma_F \sim \sigma'_F : (\Gamma_F)_{\mathcal{X}} \to \emptyset$ using rule TLR[1]. By definition of F, and since the refinement types in Γ only contain ground terms by assumption, we also know that $\overline{\Gamma}$ does not contain refinement types. Hence, by Lemma A.22, and Lemma A.10, there exists $c_\phi \subseteq [\![c]\!]_{\sigma_F,\sigma'_F}$, *i.e.* $c_\phi \subseteq \overline{c}$ such that $\overline{\Gamma} \vdash \phi_\ell(c)\sigma_F \sim \phi_r(c)\sigma'_F : \mathtt{LL} \to c_\phi$. Since $\overline{c} = [\![c]\!]_{\sigma_F,\sigma'_F}$, this proves that $\overline{\Gamma} \vdash \phi_\ell(\overline{c}) \sim \phi_r(\overline{c}) : \mathtt{LL} \to c_\phi$. Besides, it is clear from the definition of $\phi^{\Gamma}_{\mathtt{LL}}$ and rules TCstFN, TNonceL, TKey, TPubKeyL, TVKeyL that $\overline{\Gamma} \vdash \phi^{\Gamma}_{\mathtt{LL}} \sim \phi^{\Gamma}_{\mathtt{LL}} : \mathtt{LL} \to \emptyset$.

These two results prove the lemma. □

Lemma A.35. *For all ground σ, σ' such that $\exists \Gamma' \subseteq \overline{\Gamma}.\ \exists c_\sigma \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}.\ \Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c_\sigma$, there exists $c_\phi \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}$ such that $\overline{\Gamma} \vdash \phi^{\Gamma}_{\mathtt{LL}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}) \sim \phi^{\Gamma}_{\mathtt{LL}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}) : \mathtt{LL} \to c_\phi$.*

Proof. By Lemma A.34, there exists $c'_\phi \subseteq \overline{c}$ such that $\overline{\Gamma} \vdash \phi^{\Gamma}_{\mathtt{LL}} \cup \phi_\ell(\overline{c}) \sim \phi^{\Gamma}_{\mathtt{LL}} \cup \phi_r(\overline{c}) : \mathtt{LL} \to c'_\phi$. Since $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c_\sigma$, and since $(\overline{\Gamma} \backslash \Gamma'_{\mathcal{X}})$ does not contain variables with finite nonce types, by Lemma A.22, there exists $c_\phi \subseteq \left[\!\left[c'_\phi\right]\!\right]_{\sigma,\sigma'} \cup c_\sigma$ such that $\overline{\Gamma} \backslash \Gamma'_{\mathcal{X}} \vdash (\phi^{\Gamma}_{\mathtt{LL}} \cup \phi_\ell(\overline{c}))\sigma \sim (\phi^{\Gamma}_{\mathtt{LL}} \cup \phi_r(\overline{c}))\sigma' : \mathtt{LL} \to c_\phi$.

Hence, by Lemma A.10, we have $\overline{\Gamma} \vdash \phi^{\Gamma}_{\mathtt{LL}} \cup \phi_\ell(\overline{c})\sigma \sim \phi^{\Gamma}_{\mathtt{LL}} \cup \phi_r(\overline{c})\sigma' : \mathtt{LL} \to c_\phi$.

In addition, $c'_\phi \subseteq \overline{c}$ and $c_\sigma \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}$, we have $\left[\!\left[c'_\phi\right]\!\right]_{\sigma,\sigma'} \cup c_\sigma \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}$. Therefore $c_\phi \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}$, which concludes the proof. □

Lemma A.36. *For all ground σ, σ', for all recipe R such that*

- $\exists \Gamma' \subseteq \overline{\Gamma}.\ \exists c_\sigma \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}.\ \Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c_\sigma$,
- $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi^{\overline{\Gamma}}_{\mathtt{LL}} \cup \phi_\ell(\overline{c}))$,
- $R(\phi^{\overline{\Gamma}}_{\mathtt{LL}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow \neq \bot$ *and* $R(\phi^{\overline{\Gamma}}_{\mathtt{LL}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow \neq \bot$,
- $\phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})$ *and* $\phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})$ *restricted to* $\mathrm{vars}(R)$ *are ground,*

there exists a recipe R' without destructors, i.e. *in which* `dec`, `adec`, `checksign`, π_1, π_2, *do not appear, such that*

- $\mathrm{vars}(R') \subseteq \mathrm{vars}(R)$,

- $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$,
- $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))$.

Proof. We prove the property by induction on R.

- If $R = x \in \mathcal{AX}$ or $R = a \in \mathcal{C} \cup \mathcal{FN}$ then the claim holds with $R' = R$.
- If the head symbol of R is a constructor, *i.e.* if there exist R_1, R_2 such that $R = \mathtt{pk}(R_1)$ or $R = \mathtt{vk}(R_1)$ or $R = \mathtt{enc}(R_1, R_2)$ or $R = \mathtt{aenc}(R_1, R_2)$ or $R = \mathtt{sign}(R_1, R_2)$ or $R = \langle R_1, R_2 \rangle$ or $R = \mathtt{h}(R_1)$, we may apply the induction hypothesis to R_1 (and R_2 when it is present). All these case are similar, we write the proof generically for $R = f(R_1, R_2)$. By the induction hypothesis, there exist R_1', R_2' such that
  - for all $i \in \{1, 2\}$, $\mathrm{vars}(R_i') \subseteq \mathrm{vars}(R_i)$,
  - for all $i \in \{1, 2\}$, $R_i(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = R_i'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$,
  - for all $i \in \{1, 2\}$, $R_i(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = R_i'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))$.

  Let $R' = f(R_1', R_2')$. The first point imply that R' satisfies the conditions on variables. Since $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = f(R_1(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow, R_2(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow)$, the second point implies that $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$. Similarly, $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))$, and the claim holds.
- If $R = \mathtt{dec}(S, K)$ for some recipes S, K, then since $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow \neq \bot$, we have

  $$K(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = k$$

  for some $k \in K$, and $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = \mathtt{enc}(M, k)$, where $M = R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow$.

  Similarly, there exists $k' \in \mathcal{K}$ such that $K(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = k'$ and $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = \mathtt{enc}(N, k')$, where $N = R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow$.

  In addition, by Lemma A.35, there exists c' such that $\overline{\Gamma} \vdash \phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}) \sim \phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}) : \mathtt{LL} \to c'$. Thus by Lemma A.21, there exists c'' such that $\overline{\Gamma} \vdash K(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow \sim K(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow : \mathtt{LL} \to c''$, which is to say $\overline{\Gamma} \vdash k \sim k' : \mathtt{LL} \to c''$. Hence by Lemma A.18 and by well-formedness of Γ, $k = k'$ and $\Gamma(k, k) <: \mathrm{key}^{\mathtt{LL}}(T)$ for some type T.

  Since $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = \mathtt{enc}(M, k) \neq \bot$, and $\mathrm{vars}(S) \subseteq \mathrm{vars}(R)$, by the induction hypothesis, there exists S' such that $\mathrm{vars}(S') \subseteq \mathrm{vars}(S)$, $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = \mathtt{enc}(M, k)$, and $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})) = S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = \mathtt{enc}(N, k)$.

  It is then clear that either $S' = x$ for some variable $x \in \mathcal{AX}$, or $S' = \mathtt{enc}(S'', K')$ for some S'', K'. We have already shown that $\Gamma(k, k) <: \mathrm{key}^{\mathtt{LL}}(T)$. In addition, by Lemma A.33, $\mathtt{step2}_{\overline{\Gamma}}([\![\overline{c}]\!]_{\sigma,\sigma'}) = \mathtt{true}$. Hence $[\![\overline{c}]\!]_{\sigma,\sigma'}$ only contains messages encrypted with keys k'' such that $\Gamma(k'', k''') <: \mathrm{key}^{\mathtt{HH}}(T')$ or $\Gamma(k''', k'') <: \mathrm{key}^{\mathtt{HH}}(T')$ for some T', k'''. Therefore the first case is not possible.

  Hence there exist S'', K' such that $S' = \mathtt{enc}(S'', K')$. Since $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = \mathtt{enc}(M, k)$, we have $S''(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = M$. Hence $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\!\downarrow = M =$

$S''(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$, and similarly for $\phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})$. Moreover, S'', being a subterm of S', also satisfies the conditions on the domains, and thus the property holds with $R' = S''$.

- If $R = \mathtt{adec}(S, K)$ for some recipes S, K: this case is similar to the symmetric case.

- If $R = \mathtt{checksign}(S, K)$ for some recipes S, K: then since $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow \neq \bot$, we have $K(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \mathtt{vk}(k)$ for some $k \in K$, and $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \mathtt{sign}(M, k)$, where $M = R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow$.

  Similarly, there exists $k' \in \mathcal{K}$ such that $K(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \mathtt{vk}(k')$ and $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \mathtt{sign}(N, k')$, where $N = R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow$.

  Since $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \mathtt{sign}(M, k) \neq \bot$, by the induction hypothesis, there exists S' such that $\mathrm{vars}(S') \subseteq \mathrm{vars}(S)$, $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \mathtt{sign}(M, k)$, and $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})) = S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \mathtt{sign}(N, k)$.

  Since $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = \mathtt{sign}(M, k)$, it is clear from the definition of $\downarrow$ that either $S' = x$ for some $x \in \mathcal{AX}$, or $S' = \mathtt{sign}(S'', K')$ for some S'', K'.

  - In the first case, we therefore have $\mathtt{sign}(M, k) \sim \mathtt{sign}(N, k') \in [\![\overline{c}]\!]_{\sigma,\sigma'}$. In addition, by Lemma A.35, there exists $c' \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}$ such that $\overline{\Gamma} \vdash \phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}) \sim \phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}) : \mathtt{LL} \to c'$. Thus there exists $c'' \subseteq c'$ such that $\overline{\Gamma} \vdash \mathtt{sign}(M, k) \sim \mathtt{sign}(N, k') : \mathtt{LL} \to c''$. Hence by Lemma A.16, there exists $c''' \subseteq c'' \subseteq c'$ such that $\overline{\Gamma} \vdash M \sim N : \mathtt{LL} \to c'''$. Moreover M, N are ground, since by assumption $\phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})$ and $\phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})$ restricted to $\mathrm{vars}(R)$ are ground. Therefore, by Lemma A.24, there exists a recipe R' without destructors such that $M = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell(c'''))$ and $N = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r(c'''))$. Since $c''' \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}$, this proves the claim for this case.
  - In the second case, there exist S'', K' such that $S' = \mathtt{sign}(S'', K')$. Since $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = \mathtt{sign}(M, k)$, we have $S''(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = M$. Hence $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = M = S''(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$, and similarly for $\phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})$. Moreover, S'' being a subterm of S' it also satisfies the conditions on the domains, and thus the property holds with $R' = S''$.

- If $R = \pi_1(S)$ for some recipe S then since $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow \neq \bot$, we have $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \langle M_1, M_2\rangle$, where $M_1 = R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow$, and M_2 is a message.

  Similarly, $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \langle N_1, N_2\rangle$, where $N_1 = R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow$, and N_2 is a message.

  Since $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \langle M_1, M_2\rangle \neq \bot$, by the induction hypothesis, there exists S' such that $\mathrm{vars}(S') \subseteq \mathrm{vars}(S)$, $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \langle M, k\rangle$, and $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})) = S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = \langle N, k\rangle$.

  Since $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = \langle M_1, M_2\rangle$, it is clear from the definition of $\downarrow$ that either $S' = x$ for some $x \in \mathcal{AX}$, or $S' = \langle S_1, S_2\rangle$ for some S_1, S_2.

  The first case is impossible, since by Lemma A.33, $\mathtt{step2}_{\overline{\Gamma}}([\![\overline{c}]\!]_{\sigma,\sigma'}) = \mathtt{true}$, and thus $[\![\overline{c}]\!]_{\sigma,\sigma'}$ does not contain pairs.

In the second case, there exist S_1, S_2 such that $S' = \langle S_1, S_2 \rangle$. Since $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = \langle M_1, M_2 \rangle$, we have $S_1(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = M_1$. Hence $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = M_1 = S_1(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$, and similarly for $\phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})$. Moreover, S_1, being a subterm of S', also satisfies the conditions on the domains, and thus the property holds with $R' = S_1$.

- If $R = \pi_2(S)$ for some S: this case is similar to the π_1 case.

□

Lemma A.37. *For all term t and substitution σ containing only messages, if $t\downarrow \neq \bot$, then $(t\sigma)\downarrow = (t\downarrow)\sigma$.*

Proof. This property is easily proved by induction on t. In the base case where t is a variable x, by definition of $\downarrow$, since $\sigma(x)$ is a messages, $\sigma(x)\downarrow = \sigma(x)$ and the claim holds. In the other base cases where t is a name, key or constant the claim trivially holds. We prove the case where t starts with a constructor other than `enc`, `aenc`, `sign` generically for $t = f(t_1, t_2)$. We then have $t_1\sigma\downarrow \neq \bot$ and $t_2\sigma\downarrow \neq \bot$, and $t\sigma\downarrow = f(t_1\sigma\downarrow, t_2\sigma\downarrow)$, which, by the induction hypothesis, is equal to $f(t_1\downarrow \sigma, t_2\downarrow \sigma)$, *i.e.* to $f(t_1, t_2)\downarrow \sigma$. The case where f is `enc`, `aenc` or `sign` is similar, but we in addition know that $t_2\downarrow$ is a key.

Finally if t starts with a destructor, $t = d(t_1, t_2)$, we know that $t_1\downarrow$ starts with the corresponding constructor f: $t_1\downarrow = f(t_3, t_4)$. In the case of encryptions and signatures we know in addition that $t_4\downarrow$ and $t_2\downarrow$ are the same key (resp. public key/verification key). We then have $t\downarrow = t_3\downarrow$, and $t\sigma\downarrow = t_3\sigma\downarrow$ (or t_4 in the case of the second projection π_2). Hence by the induction hypothesis, $t\sigma\downarrow = t_3\downarrow \sigma = t\downarrow \sigma$ and the claim holds.

□

Lemma A.38. *For all ground σ, σ', for all recipe R such that*

- $\exists \Gamma' \subseteq \overline{\Gamma}.\ \exists c_\sigma.\ \Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c_\sigma$,
- $\mathrm{vars}(R) \subseteq \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell(\overline{c}))$,
- *$\phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})$ and $\phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})$ restricted to $\mathrm{vars}(R)$ are ground,*

for all $x \in \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell(\overline{c}))$, if $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = (\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))(x)$ then R is a variable $y \in \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell(\overline{c}))$, or $R \in \mathcal{C} \cup \mathcal{FN}$.

Similarly, if $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}) = (\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))(x)$ then R is a variable $y \in \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r(\overline{c}))$ or $R \in \mathcal{C} \cup \mathcal{FN}$.

Proof. We only detail the proof for $\phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})$, as the proof for $\phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})$ is similar.

We distinguish several cases for R.

- If $R = a \in \mathcal{C} \cup \mathcal{FN}$: the claim clearly holds.
- If $R = x \in \mathcal{AX}$ then the claim trivially holds.
- If $R = \mathtt{enc}(S, K)$ or $\mathtt{sign}(S, K)$ for some recipes S, K: these two cases are similar, we only detail the encryption case. $(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))(x)$ is then an encrypted message. By Lemma A.33, $\mathtt{step2}_{\overline{\Gamma}}([\![\overline{c}]\!]_{\sigma,\sigma'}) = \mathtt{true}$. Hence there exist $k, k' \in \mathcal{K}$ and T such that $K(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = k$ and $\Gamma(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$. This is only possible if there exists a

variable z such that $K = z$ and $(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\tilde{c}]\!]_{\sigma,\sigma'}))(z) = k$. Since $\mathtt{step2}_{\overline{\Gamma}}(\overline{c}) = \mathtt{true}$, z can only be in $\mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}})$. By definition of $\phi_{\mathrm{LL}}^{\overline{\Gamma}}$, this implies that $\Gamma(k,k) <: \mathrm{key}^{\mathrm{LL}}(T')$ for some T'. Since $\overline{\Gamma}$ is well-formed, this contradicts $\Gamma(k,k') <: \mathrm{key}^{\mathrm{HH}}(T)$: this case is impossible.

- If $R = \mathtt{aenc}(S,K)$ or $\mathtt{h}(S)$ for some recipes S, K: these two cases are similar, we only detail the encryption case. $(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))(x)$ is then an asymmetrically encrypted message. By Lemma A.33, we know that $\mathtt{step2}_{\overline{\Gamma}}([\![\overline{c}]\!]_{\sigma,\sigma'}) = \mathtt{true}$. Hence $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$ contains directly under pairs a nonce n such that $\overline{\Gamma}(n) = \tau_n^{\mathrm{HH},a}$, or a key $k \in \mathcal{K}$ such that $\overline{\Gamma}(k,k') = \mathrm{key}^{\mathrm{HH}}(T)$ for some T, k'.

  This is only possible if there exists a recipe S' such that $S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = n$ (resp. k). Since S' can only contain names from $\mathcal{FN}$ (and no keys), this implies that there exists a variable z such that $(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))(z) = n$ (resp. k). As $\mathtt{step2}_{\overline{\Gamma}}(\overline{c}) = \mathtt{true}$, z can only be in $\mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}})$. Thus, by definition of $\phi_{\mathrm{LL}}^{\overline{\Gamma}}$, $\overline{\Gamma}(n) = \tau_n^{\mathrm{LL},a}$ for some a (resp. $\overline{\Gamma}(k,k) <: \mathrm{key}^{\mathrm{LL}}(T')$ for some T'), which is contradictory (as $\overline{\Gamma}$ is well-formed).

- Finally, the head symbol of R cannot be $\langle\cdot,\cdot\rangle$, $\mathtt{dec}$, $\mathtt{adec}$, $\mathtt{checksign}$, π_1, π_2 since $\mathtt{step2}_{\overline{\Gamma}}([\![\overline{c}]\!]_{\sigma,\sigma'}) = \mathtt{true}$ by Lemma A.33.

□

Lemma A.39. *For all ground σ, σ', for all recipes R, S such that*

- $\exists \Gamma' \subseteq \overline{\Gamma}.\ \exists c_\sigma \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}.\ \Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c_\sigma$,
- $\mathrm{vars}(R) \cup \mathrm{vars}(S) \subseteq \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell(\overline{c}))$,
- $\phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})$ *and* $\phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})$ *restricted to* $\mathrm{vars}(R) \cup \mathrm{vars}(S)$ *are ground,*

we have

$$(R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))){\downarrow} = (S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))){\downarrow} \iff (R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))){\downarrow} = (S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))){\downarrow}\ .$$

Proof. We only detail the proof for the $(\Rightarrow)$ direction, as the other direction is similar. We then assume that

$$(R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))){\downarrow} = (S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))){\downarrow}\ .$$

Let us first note that

$$(R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))){\downarrow} \neq \bot \quad \iff \quad (R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))){\downarrow} \neq \bot.$$

This follows from Lemmas A.35 and A.21.

Similarly, we have

$$S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} \neq \bot \quad \iff \quad S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} \neq \bot.$$

Therefore, if $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} = \bot$, then $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} = \bot$. By assumption, in that case we also have $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} = \bot$, and thus $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} = \bot$, and the claim holds.

Let us now assume that $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} \neq \bot$, *i.e.*, by assumption, that $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} \neq \bot$. We then have $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} \neq \bot$ and $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'})){\downarrow} \neq \bot$.

By Lemma A.36, there exist recipes R', S' without destructors such that

- $\mathrm{vars}(R') \cup \mathrm{vars}(S') \subseteq \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$,
- $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))\downarrow = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$,
- $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))\downarrow = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))$.
- $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))\downarrow = S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$,
- $S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))\downarrow = S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))$.

By the assumption, we have $R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})) = S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$.

We show that $R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'})) = S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))$ holds, by induction on the recipes R' and S'. Since $R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})) = S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$, we can distinguish four cases for R' and S'.

- If they have the same head symbol, either this symbol is a nonce or constant and the claim is trivial, or it is a variable, and we handle this case later, or it is a constructor. We write the proof for this last case generically for $R' = f(R'')$ and $S' = f(S'')$. We have necessarily $R''(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})) = S''(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$. It then follows by applying the induction hypothesis to R'' and S'' that $R''(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'})) = S''(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))$. The claim follows by applying f on both sides of this equality.
- If R' is a variable and not S': then by Lemma A.38, $S' \in \mathcal{C} \cup \mathcal{FN}$. Let us denote $R' = x$. By Lemma A.35, there exists c_x such that $\overline{\Gamma} \vdash (\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))(x) \sim (\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))(x) : \mathtt{LL} \to c_x$. Since $(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))(x) = R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})) = S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})) \in \mathcal{C} \cup \mathcal{FN}$, by Lemma A.18, we have $(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))(x) = S'(\phi_{\mathrm{LL}}^{\Gamma} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$, *i.e.* $R'(\phi_{\mathrm{LL}}^{\Gamma} \cup \phi_r(\tilde{c})) = S'(\phi_{\mathrm{LL}}^{\Gamma} \cup \phi_r(\tilde{c}))$.
- If S' is a variable and not R': this case is similar to the previous one.
- If R', S' are two variables x and y, we have $(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))(x) = (\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))(y)$. We can then prove $(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))(x) = (\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\bar{c}]\!]_{\sigma,\sigma'}))(y)$. Indeed:
  - if $x, y \in \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}})$, this follows from the definition of $\phi_{\mathrm{LL}}^{\overline{\Gamma}}$.
  - if $x \in \mathrm{dom}(\phi_{\mathrm{LL}}^{\overline{\Gamma}})$ and $y \in \mathrm{dom}(\phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$: then by definition of $\phi_{\mathrm{LL}}^{\overline{\Gamma}}$, $R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})) = \phi_{\mathrm{LL}}^{\overline{\Gamma}}(x)$ is a nonce, key, public key, or verification key. Hence $\phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})(y)$ is also a nonce, key, public key or verification key. This is not possible, as by Lemma A.33, $\mathtt{step2}_{\overline{\Gamma}}([\![\bar{c}]\!]_{\sigma,\sigma'}) = \mathtt{true}$.
  - if $x, y \in \mathrm{dom}(\phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'}))$: then there exist $M \sim M' \in \bar{c}$, $N \sim N' \in \bar{c}$ such that $\phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})(x) = M\sigma$, $\phi_r([\![\bar{c}]\!]_{\sigma,\sigma'})(x) = M'\sigma'$, $\phi_\ell([\![\bar{c}]\!]_{\sigma,\sigma'})(y) = N\sigma$, $\phi_r([\![\bar{c}]\!]_{\sigma,\sigma'})(y) = N'\sigma'$. Since $M\sigma = N\sigma$, M, N are unifiable, let μ be their most general unifier. There exists θ such that $\sigma = \mu\theta$.

    Let then α be the restriction of μ to $\{x \in \mathrm{vars}(M) \cup \mathrm{vars}(N) \mid \overline{\Gamma}(x) = \mathtt{LL} \wedge \mu(x) \in \mathcal{N} \text{ is a nonce}\}$.

    By $\mathtt{step3}_{\overline{\Gamma}}(\bar{c})$, we have $M'\alpha = N'\alpha$.

Note that α and β have disjoint domains. Let $x \in \mathrm{dom}(\alpha) \cap (\mathrm{vars}(M') \cup \mathrm{vars}(N'))$. Then $\mu(x) = n$ for some $n \in \mathcal{N}$. Thus $\sigma(x) = \mu(x)\theta = n$.
Since, by assumption, $\exists \Gamma' \subseteq \overline{\Gamma}.\ \exists c_\sigma.\ \Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c_\sigma$, there exists c_x such that $\Gamma'_{\mathcal{N},\mathcal{K}} \vdash n \sim \sigma'(x) : \mathtt{LL} \to c_x$. Hence by Lemma A.18, $\sigma'(x) = n = \mu(x) = \alpha(x) = (\alpha\sigma')(x)$.
We have shown that for all $x \in \mathrm{dom}(\alpha) \cap (\mathrm{vars}(M') \cup \mathrm{vars}(N'))$, $(\alpha\sigma')(x) = \sigma'(x)$. Thus, on $\mathrm{dom}(\alpha) \cap (\mathrm{vars}(M') \cup \mathrm{vars}(N'))$, we have $\alpha\sigma' = \sigma'$.

Therefore, since $M'\alpha = N'\alpha$, we have $M'\alpha\sigma' = N'\alpha\sigma'$, *i.e.* $M'\sigma' = N'\sigma'$. This proves the claim.

Finally, since $R'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'})) = S'(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![\overline{c}]\!]_{\sigma,\sigma'}))$, we have $R(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow = S(\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![\overline{c}]\!]_{\sigma,\sigma'}))\downarrow$.
This proves the property. □

Lemma A.40. *For all ground σ, σ', for all $\Gamma' \subseteq \overline{\Gamma}$, and for all $c' \subseteq \overline{c}$, such that*

- $\exists c_\sigma \subseteq [\![\overline{c}]\!]_{\sigma,\sigma'}.\ \Gamma'_{\mathcal{N},\mathcal{K}} \vdash \sigma \sim \sigma' : \Gamma'_{\mathcal{X}} \to c_\sigma$,
- $\Gamma'_{\mathcal{N},\mathcal{K}} = \overline{\Gamma}_{\mathcal{N},\mathcal{K}}$ *and* $\mathrm{vars}(c') \subseteq \mathrm{dom}(\Gamma')$,

the frames $\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_\ell([\![c']\!]_{\sigma,\sigma'})$ and $\phi_{\mathrm{LL}}^{\overline{\Gamma}} \cup \phi_r([\![c']\!]_{\sigma,\sigma'})$ are statically equivalent.

Proof. This is a direct consequence of Lemma A.39, by unfolding the definition of static equivalence. □

Lemma A.41. *c is consistent in Γ.*

Proof. By Lemma A.32, it suffices to show that $\overline{c}$ is consistent in $\overline{\Gamma}$. This is a direct consequence of Lemma A.40, by unfolding the definition of consistency. □

Theorem A.6 (Soundness of the procedure)**.** *Let C be a constraint set without infinite nonce types,* i.e.

$$\forall (c, \Gamma) \in C.\ \forall l, l', m, n. \Gamma(x) \neq [\![\tau_m^{l,\infty}\ ;\ \tau_n^{l',\infty}]\!].$$

If `check_const`(C) *succeeds, then C is consistent.*

Proof. The previous lemmas directly imply that for all $(c, \Gamma) \in C$, c is consistent in Γ. This proves the theorem. □

A.4 Consistency for replicated processes

In this section, we prove the results regarding the procedure when checking consistency in the replicated case. We recall that a table displaying the correspondence between the numbering of the Lemmas and Theorems in Chapter 2 and here is provided at the beginning of Appendix A.

In this section, we only consider constraints obtained by typing processes (with the same key types). Notably, by the well-formedness assumptions on the processes, this means that a nonce n is always associated with the same nonce type.

Theorem A.7. *Let C and C' be two constraint sets that*

- *do not share any common variable,* i.e.

$$\forall (c,\Gamma) \in C.\ \forall (c',\Gamma') \in C'.\ \mathrm{dom}(\Gamma_{\mathcal{X}}) \cap \mathrm{dom}(\Gamma'_{\mathcal{X}}) = \emptyset;$$

- *only share nonces which have the same finite nonce type,* i.e.

$$\forall (c,\Gamma) \in C.\ \forall (c',\Gamma') \in C'.\ \forall m \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') \cap \mathcal{N}.\ \exists l.\ \Gamma(m) = \Gamma'(m) = \tau_m^{l,1};$$

- *only share keys which are paired in the same way and have the same types,* i.e.

$$\forall (c,\Gamma) \in C.\ \forall (c',\Gamma') \in C'.\ \forall k \in \mathrm{keys}(\Gamma) \cap \mathrm{keys}(\Gamma').\forall k' \in \mathcal{K}.$$
$$((k,k') \in \mathrm{dom}(\Gamma) \Leftrightarrow (k,k') \in \mathrm{dom}(\Gamma')) \ \wedge\ ((k',k) \in \mathrm{dom}(\Gamma) \Leftrightarrow (k',k) \in \mathrm{dom}(\Gamma'))$$

and

$$\forall (c,\Gamma) \in C.\ \forall (c',\Gamma') \in C'.\ \forall (k,k') \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma').\ \Gamma(k,k') = \Gamma'(k,k').$$

For all $n \in \mathbb{N}$, *if* $\texttt{check\_const}([\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n) = \texttt{true}$, *then*

$$\texttt{check\_const}(\cup_{\times 1 \le i \le n} [\,C\,]_i^n) \cup_\times [\,C'\,]_1^n) = \texttt{true}.$$

Proof. Let $n \in \mathbb{N}$. Let C, C' be as defined in the statement of the theorem.

Let $(c,\Gamma) \in (\cup_{\times 1 \le i \le n} [\,C\,]_i^n) \cup_\times [\,C'\,]_1^n$. By definition of $\cup_\times$, there exists $(c',\Gamma') \in [\,C'\,]_1^n$, and for all $i \in [\![1,n]\!]$, there exists $(c_i,\Gamma_i) \in [\,C\,]_i^n$, such that

- $c = (\cup_{1 \le i \le n} c_i) \cup c'$;
- $\Gamma = (\bigcup_{1 \le i \le n} \Gamma_i) \cup \Gamma'$.

Let $i \in [\![1,n]\!]$. Since $(c_i,\Gamma_i) \in [\,C\,]_i^n$, by definition of $[\,C\,]_i^n$ there exists $(c'_i,\Gamma'_i) \in C$ such that

- $c_i = [\,c'_i\,]_i^{\Gamma_i}$;
- $\Gamma_i \in \mathrm{branches}([\,\Gamma'_i\,]_i^n)$.

Note that this implies $\mathrm{dom}(\Gamma_{i\mathcal{X}})$ only contains variables indexed by i, and, from the assumption that $\mathrm{vars}(c'_i) \subseteq \mathrm{dom}(\Gamma'_{i\mathcal{X}})$, that $\mathrm{vars}(c_i) \subseteq \mathrm{dom}(\Gamma_{i\mathcal{X}})$.

For all $i \in [\![1,n]\!]$, let δ_i^1 denote the function on terms which consists in exchanging all occurrences of the indices i and 1, *i.e.* replacing any occurrence of m_i (for all nonce m with an infinite nonce type) with m_1, any occurrence of m_1 with m_i, any occurrence of k_1 with k_i (for all key k), any occurrence of k_i with k_1, any occurrence of x_i with x_1 (for all variable x), and any occurrence of x_1 with x_i (also for all variable x).

We extend this function to constraints, types, typing environments and constraint sets. In the case of types we use it to denote the replacement of nonces appearing in the refinements. In the case of typing environments it denotes the replacement of nonces appearing in the types, and of nonces, keys and variables in the domain of the environment, *i.e.* $(\delta_i^1(\Gamma))(x_1) = \delta_i^1(\Gamma(x_i))$. In the case of constraint sets it denotes the application of the function to each constraint and environment in the constraint set.

Similarly, we denote δ_i^2 the function exchanging indices i and 2.

For all $h \in [\![1,n]\!]$ and all $i \neq j \in [\![1,n]\!]$, such that $i \neq 2$ and $j \neq 1$, let

$$(c_h^{i,j}, \Gamma_h^{i,j}) = (c_h, \Gamma_h)\delta_i^1\delta_j^2.$$

Similarly, for all $h \in [\![1,n]\!]$ and all $i \in [\![1,n]\!]$, let

$$(c_h^{i,i}, \Gamma_h^{i,i}) = (c_h, \Gamma_h)\delta_i^1.$$

Finally, for all $i \in [\![1,n]\!]$, let Γ'^i be the typing environment such that $\mathrm{dom}(\Gamma'^i) = \mathrm{dom}(\Gamma')$ and $\forall x \in \mathrm{dom}(\Gamma'^i).\ \Gamma'^i(x) = \Gamma'(x)\delta_i^1$.

Since $(c_i, \Gamma_i) \in [\![C]\!]_i^n$, we can show that $(c_i^{i,j}, \Gamma_i^{i,j}) \in [\![C]\!]_1^n$. Indeed, recall that there exists $(c'_i, \Gamma'_i) \in C$ such that $c_i = [\![c'_i]\!]_i^{\Gamma_i}$ and $\Gamma_i \in \mathrm{branches}([\![\Gamma'_i]\!]_i^n)$. c_i only contains variables, keys and names indexed by i or unindexed, hence it is clear that $c_i^{i,j} = [\![c'_i]\!]_1^{\Gamma_i}$. Moreover, since $\Gamma_i \in \mathrm{branches}([\![\Gamma'_i]\!]_i^n)$, it is clear that $\Gamma_i^{i,j} \in \mathrm{branches}([\![\Gamma'_i]\!]_i^n\delta_i^1\delta_j^2)$. By definition, indexed nonces, variables, or keys appear in $[\![\Gamma'_i]\!]_i^n$ only in its domain, and as parts of union types of the form $[\![\tau_{m_1}^{l,1}\,;\,\tau_{p_1}^{l',1}]\!] \vee \ldots \vee [\![\tau_{m_n}^{l,1}\,;\,\tau_{p_n}^{l',1}]\!]$. This union type is left unchanged by $\delta_i^1\delta_j^2$: since $i \neq j$, $i \neq 2$, and $j \neq 1$, $\delta_i^1\delta_j^2$ is indeed only performing a permutation of the indices. Hence, $[\![\Gamma'_i]\!]_i^n\delta_i^1\delta_j^2 = [\![\Gamma'_i]\!]_1^n$. Thus $\Gamma_i^{i,j} \in \mathrm{branches}([\![\Gamma'_i]\!]_1^n)$. Therefore, $(c_i^{i,j}, \Gamma_i^{i,j}) \in [\![C]\!]_1^n$.

Note that $\mathrm{dom}(\Gamma_i^{i,j})$ only contains variables indexed by 1; and that, since $\mathrm{vars}(c_i) \subseteq \mathrm{dom}(\Gamma_i)$, we have $\mathrm{vars}(c_i^{i,j}) \subseteq \mathrm{dom}(\Gamma_i^{i,j})$.

Similarly, if $j \neq i$, $i \neq 2$ and $j \neq 1$, $(c_j^{i,j}, \Gamma_j^{i,j}) \in [\![C]\!]_2^n$. Note that $\mathrm{dom}(\Gamma_j^{i,j})$ only contains variables indexed by 2; and that $\mathrm{vars}(c_j^{i,j}) \subseteq \mathrm{dom}(\Gamma_j^{i,j})$.

Similarly, we also have $(c', \Gamma'^i) \in [\![C']\!]_1^n$.

By assumption, for all $(c, \Gamma) \in C$ and for all $(c', \Gamma') \in C'$, $\mathrm{dom}(\Gamma_{\mathcal{X}}) \cap \mathrm{dom}(\Gamma'_{\mathcal{X}}) = \emptyset$, and Γ, Γ' give the same types to the nonces and keys they have in common. Hence for all $(c'', \Gamma'') \in [\![C]\!]_1^n$, and all $(c''', \Gamma''') \in [\![C']\!]_1^n$, we know that Γ'' and Γ''' are compatible. In particular this applies to all the $\Gamma_i^{i,j}$ and Γ' (as well as $\Gamma_i^{i,j}$ and Γ'^i).

Moreover, for all $(c'', \Gamma'') \in [\![C]\!]_2^n$, and all $(c''', \Gamma''') \in [\![C']\!]_1^n$, since $\mathrm{dom}(\Gamma''') \subseteq \{x_1 \mid x \in \mathcal{X}\}$, and $\mathrm{dom}(\Gamma'') \subseteq \{x_2 \mid x \in \mathcal{X}\}$, Γ'' and Γ''' are also compatible. This in particular applies to $\Gamma_j^{i,j}$ for all $i \neq j \in [\![1,n]\!]$ and Γ' (as well as $\Gamma_j^{i,j}$ and Γ'^i).

If C is empty, then so is $\cup_{\times 1 \leq i \leq n} [\![C]\!]_i^n$ and the claim clearly holds. Let us now assume that C is not empty. Hence for all $i \in [\![1,n]\!]$, $[\![C]\!]_i^n$ is not empty.

The procedure for c, Γ is as follows:

1. We compute $\mathtt{step1}_\Gamma(c)$. Following the notations used in the procedure, we have

   $$F = \{x \in \mathrm{dom}(\Gamma) \mid \exists m, n, l, l'.\ \Gamma(x) = [\![\tau_m^{l,1}\,;\,\tau_n^{l',1}]\!]\},$$

   and we write $(\overline{c}, \overline{\Gamma}) \stackrel{\text{def}}{=} \mathtt{step1}_\Gamma(c)$.

   For all $i \in [\![1,n]\!]$, let $(\overline{c_i}, \overline{\Gamma}_i) \stackrel{\text{def}}{=} \mathtt{step1}_{\Gamma_i}(c_i)$. Let also $(\overline{c}', \overline{\Gamma}') \stackrel{\text{def}}{=} \mathtt{step1}_{\Gamma'}(c')$. We have $\overline{c} = (\cup_{1 \leq i \leq n} \overline{c_i}) \cup \overline{c}'$, and $\overline{\Gamma} = (\cup_{1 \leq i \leq n} \overline{\Gamma}_i) \cup \overline{\Gamma}'$.

   For all $h, i, j \in [\![1,n]\!]$, such that either $i \neq j$ and $i \neq 2$ and $j \neq 1$, or $i = j$, let us also denote $(\overline{c}_h^{i,j}, \overline{\Gamma}_h^{i,j}) \stackrel{\text{def}}{=} \mathtt{step1}_{\Gamma_h^{i,j}}(c_h^{i,j})$. Similarly, for all $i \in [\![1,n]\!]$, let also $(\overline{c'}^i, \overline{\Gamma'}^i) \stackrel{\text{def}}{=} \mathtt{step1}_{\Gamma'^i}(c')$. Since, for $i \neq j$, $(c_h^{i,j}, \Gamma_h^{i,j}) = (c_h, \Gamma_h)\delta_i^1\delta_j^2$, it can easily be shown (by induction on the terms)

that $(\overline{c}_h^{i,j}, \overline{\Gamma}_h^{i,j}) = (\overline{c_h}, \overline{\Gamma}_h)\delta_i^1\delta_j^2$. Similarly, $(\overline{c}_h^{i,i}, \overline{\Gamma}_h^{i,i}) = (\overline{c_h}, \overline{\Gamma}_h)\delta_i^1$. Finally, we similarly also have $(\overline{c'}^i, \overline{\Gamma'}^i) = (\overline{c}', \overline{\Gamma}')\delta_p^1$.

2. We check that $\texttt{step2}_{\overline{\Gamma}}(\overline{c})$ holds, *i.e.* that each $M \sim N \in \overline{c}$ has the correct form (with respect to the definition of `step2`).

   If $M \sim N \in \overline{c}$, either $M \sim N \in \overline{c}'$, or there exists $i \in [\![1, n]\!]$ such that $M \sim N \in \overline{c_i}$.

   - In the first case, $M \sim N \in \overline{c}'$. By assumption, $[\, C \,]_1^n$ and $[\, C \,]_2^n$ are not empty. Hence there exist $(c'', \Gamma'') \in [\, C \,]_1^n$ and $(c''', \Gamma''') \in [\, C \,]_2^n$. Thus, $(c'' \cup c''' \cup c', \Gamma'' \cup \Gamma''' \cup \Gamma') \in [\, C \,]_1^n \cup_\times [\, C \,]_2^n \cup_\times [\, C' \,]_1^n$ (as noted previously, Γ'', Γ''', and Γ' are compatible). Hence, by assumption, $\texttt{check\_const}(\{(c'' \cup c''' \cup c', \Gamma'' \cup \Gamma''' \cup \Gamma')\})$ succeeds.
     If $\overline{c}'' = fst(\texttt{step1}_{\Gamma''}(c''))$, and $\overline{c}''' = fst(\texttt{step1}_{\Gamma'''}(c'''))$, then we have

     $$\overline{c}'' \cup \overline{c}''' \cup \overline{c}' = fst(\texttt{step1}_{\Gamma'' \cup \Gamma''' \cup \Gamma'}(c'' \cup c''' \cup c')).$$

     Therefore, $\texttt{step2}_{\overline{\Gamma}}(\tilde{c}'' \cup \tilde{c}''' \cup \tilde{c}') = \texttt{true}$.
     In particular, $M \sim N \in \overline{c}'$ has the correct form.
   - In the second case, $M \sim N \in \overline{c_i}$ for some $i \in [\![1, n]\!]$.
     Let $M' = M\delta_i^1$ and $N' = N\delta_i^1$. Since $\overline{c}_i^{i,i} = \overline{c_i}\delta_i^1$, we have $M' \sim N' \in \overline{c}_i^{i,i}$.
     By assumption, $[\, C \,]_2^n$ is not empty, hence there exists $(c'', \Gamma'') \in [\, C \,]_2^n$. Thus, $(c_i^{i,i} \cup c'' \cup c', \Gamma_i^{i,i} \cup \Gamma'' \cup \Gamma') \in [\, C \,]_1^n \cup_\times [\, C \,]_2^n \cup_\times [\, C' \,]_1^n$ (as noted previously, $\Gamma_i^{i,i}$, Γ'', and Γ' are compatible). Hence, by assumption, $\texttt{check\_const}(\{(c_i^{i,i} \cup c'' \cup c', \Gamma_i^{i,i} \cup \Gamma'' \cup \Gamma')\})$ succeeds. If $\overline{c}'' = fst(\texttt{step1}_{\Gamma''}(c''))$, then $\overline{c}_i^{i,i} \cup \overline{c}'' \cup \overline{c}' = fst(\texttt{step1}_{\Gamma''_{i,1} \cup \Gamma'' \cup \Gamma'}(c_i^{i,i} \cup c'' \cup c'))$.
     Therefore, $\texttt{step2}_{\overline{\Gamma}}(\overline{c}_i^{i,i} \cup \overline{c}'' \cup \overline{c}')$ holds.
     In particular, $M' \sim N' \in \tilde{c}_i^{i,i}$, has the correct form. By examining all the cases and using the fact that for all m_i, m_j, if m_i is associated with the type $\tau_{m_i}^{l,a}$ and m_j with $\tau_{m_j}^{l',a}$ then $l = l'$, and similarly for keys; it follows that $M \sim N$ also has the correct form.

   Therefore, $\texttt{step2}_{\overline{\Gamma}}(\overline{c})$ holds.

3. Finally, we check that $\texttt{step3}_{\overline{\Gamma}}(\overline{c})$ holds. Let $M_1 \sim N_1 \in \overline{c}$ and $M_2 \sim N_2 \in \overline{c}$. Let us prove the property in the case where M_1 and M_2 are unifiable with a most general unifier μ. The case where N_1 and N_2 are unifiable is similar.

   Let then α be the restriction of μ to $\{x \in \text{vars}(M_1) \cup \text{vars}(M_2) \mid \overline{\Gamma}(x) = \texttt{LL} \ \wedge \ \mu(x) \in \mathcal{N}\}$.

   We have to prove that $N_1\alpha = N_2\alpha$.

   Since we already have $\overline{c} = (\cup_{1 \leq i \leq n} \overline{c_i}) \cup \overline{c}'$, we know that:

   - either there exists $i \in [\![1, n]\!]$ such that $M_1 \sim N_1 \in \overline{c_i}$;
   - or $M_1 \sim N_1 \in \overline{c}'$;

   and

   - either there exists $j \in [\![1, n]\!]$ such that $M_2 \sim N_2 \in \overline{c_j}$;

- or $M_2 \sim N_2 \in \overline{c}'$.

Let us first prove the case where there exist $i, j \in [\![1, n]\!]$ such that $M_1 \sim N_1 \in \overline{c_i}$ and $M_2 \sim N_2 \in \overline{c_j}$. We distinguish two cases.

- if $i \neq j$: The property to prove is symmetric between $M_1 \sim N_1 \in \overline{c}$ and $M_2 \sim N_2 \in \overline{c}$. Hence without loss of generality we may assume that $i \neq 2$ and $j \neq 1$.
  Let then $M_1' = M_1\delta_i^1\delta_j^2$, $N_1' = N_1\delta_i^1\delta_j^2$, $M_2' = M_2\delta_i^1\delta_j^2$, $N_2' = N_2\delta_i^1\delta_j^2$.
  Since $\overline{c}_i^{i,j} = \overline{c_i}\delta_i^1\delta_j^2$, we have $M_1' \sim N_1' \in \overline{c}_i^{i,j}$. Similarly, $M_2' \sim N_2' \in \overline{c}_j^{i,j}$.
  Since M_1 and M_2 are unifiable, then so are M_1' and M_2', with a most general unifier μ' which satisfies $\mu(x) = t \Leftrightarrow \mu'(x\delta_i^1\delta_j^2) = t\delta_i^1\delta_j^2$.
  Let then α' be the restriction of μ' to $\{x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2') \mid (\overline{\Gamma}_i^{i,j} \cup \overline{\Gamma}_j^{i,j})(x) = \mathtt{LL} \wedge \mu'(x) \in \mathcal{N} \text{ is a nonce}\}$.
  Similarly α' is such that $\forall x \in \mathrm{dom}(\alpha').\forall n.\ \alpha(x) = n \Leftrightarrow \alpha'(x\delta_i^1\delta_j^2) = n\delta_i^1\delta_j^2$, *i.e.* $\delta_i^1\delta_j^2\alpha'\delta_i^1\delta_j^2 = \alpha$.
  By assumption, $\mathtt{check\_const}(\{(c_i^{i,j} \cup c_j^{i,j} \cup c', \Gamma_i^{i,j} \cup \Gamma_j^{i,j} \cup \Gamma')\})$ succeeds since $(c_i^{i,j} \cup c_j^{i,j} \cup c', \Gamma_i^{i,j} \cup \Gamma_j^{i,j} \cup \Gamma') \in [\, C \,]_1^n \cup_\times [\, C \,]_2^n \cup_\times [\, C' \,]_1^n$.
  Thus, $\mathtt{step3}_{\overline{\Gamma}_i^{i,j} \cup \overline{\Gamma}_j^{i,j} \cup \overline{\Gamma}'}(\overline{c}_i^{i,j} \cup \overline{c}_j^{i,j} \cup \overline{c}')$ holds, and since $\{M_1' \sim N_1', M_2' \sim N_2'\} \subseteq \overline{c}_i^{i,j} \cup \overline{c}_j^{i,j} \cup \overline{c}'$, we know that since M_1', M_2' are unifiable, $N_1'\alpha' = N_2'\alpha'$.
  Thus $N_1'\alpha'\delta_i^1\delta_j^2 = N_2'\alpha'\delta_i^1\delta_j^2$, *i.e.*, since $i \neq j$, and $\delta_i^1\delta_j^2\alpha'\delta_i^1\delta_j^2 = \alpha$, $N_1\alpha = N_2\alpha$. Therefore the claim holds in this case.
- if $i = j$ then let $M_1' = M_1\delta_i^1$, $N_1' = N_1\delta_i^1$, $M_2' = M_2\delta_i^1$, $N_2' = N_2\delta_i^1$. Since $\overline{c}_i^{i,i} = \overline{c_i}\delta_i^1$, we have $M_1' \sim N_1' \in \overline{c}_i^{i,i}$. Similarly, $M_2' \sim N_2' \in \overline{c}_i^{i,i}$.
  Since M_1 and M_2 are unifiable, then so are M_1' and M_2', with a most general unifier μ' which satisfies $\mu(x) = t \Leftrightarrow \mu'(x\delta_i^1) = t\delta_i^1$.
  Let then α' be the restriction of μ' to $\{x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2') \mid \overline{\Gamma}_i^{i,i}(x) = \mathtt{LL} \wedge \mu'(x) \in \mathcal{N} \text{ is a nonce}\}$.
  Similarly α' is such that $\forall x \in \mathrm{dom}(\alpha').\forall n.\ \alpha(x) = n \Leftrightarrow \alpha'(x\delta_i^1) = n\delta_i^1$, *i.e.* $\delta_i^1\alpha'\delta_i^1 = \alpha$.
  By assumption, $[\, C \,]_2^n$ is not empty, hence there exists $(c'', \Gamma'') \in [\, C \,]_2^n$. Thus, $(c_i^{i,i} \cup c'' \cup c', \Gamma_i^{i,i} \cup \Gamma'' \cup \Gamma') \in [\, C \,]_1^n \cup_\times [\, C \,]_2^n \cup_\times [\, C' \,]_1^n$. Hence, by assumption, $\mathtt{check\_const}(\{(c_i^{i,i} \cup c'' \cup c', \Gamma_i^{i,i} \cup \Gamma'' \cup \Gamma')\})$ succeeds. If $\overline{c}'' = fst(\mathtt{step1}_{\Gamma''}(c''))$, then $\overline{c}_i^{i,i} \cup \overline{c}'' \cup \overline{c}' = fst(\mathtt{step1}_{\Gamma_i^{i,i} \cup \Gamma'' \cup \Gamma'}(c_i^{i,i} \cup c'' \cup c'))$.
  Thus, $\mathtt{step3}_{\overline{\Gamma}_i^{i,i} \cup \overline{\Gamma}'' \cup \overline{\Gamma}'}(\overline{c}_i^{i,i} \cup \overline{c}'' \cup \overline{c}')$ holds, and since $\{M_1' \sim N_1', M_2' \sim N_2'\} \subseteq \overline{c}_i^{i,i} \cup \overline{c}'' \cup \overline{c}'$, we know that $N_1'\alpha' = N_2'\alpha'$.
  Thus $N_1'\alpha'\delta_i^1 = N_2'\alpha'\delta_i^1$, *i.e.*, since $\delta_i^1\alpha'\delta_i^1$, $N_1\alpha = N_2\alpha$. Therefore the claim holds in this case.

Let us now prove the case where there exists $i \in [\![1, n]\!]$ such that $M_1 \sim N_1 \in \overline{c_i}$, and $M_2 \sim N_2 \in \overline{c}'$. The symmetric case, where $M_1 \sim N_1 \in \overline{c}'$ and there exists $j \in [\![1, n]\!]$ such that $M_2 \sim N_2 \in \overline{c_j}$, is similar.

Let then $M_1' = M_1\delta_i^1$, $N_1' = N_1\delta_i^1$, $M_2' = M_2\delta_i^1$, $N_2' = N_2\delta_i^1$. Since $\overline{c}_i^{i,i} = \overline{c_i}\delta_i^1$, we have $M_1' \sim N_1' \in \overline{c}_i^{i,i}$. Similarly, $M_2' \sim N_2' \in \overline{c'}^i$.

Since M_1 and M_2 are unifiable, then so are M_1' and M_2', with a most general unifier μ' which satisfies $\mu(x) = t \Leftrightarrow \mu'(x\delta_i^1) = t\delta_i^1$.

Let then α' be the restriction of μ' to $\{x \in \text{vars}(M_1') \cup \text{vars}(M_2') \mid (\overline{\Gamma}_i^{i,i} \cup \Gamma'^i)(x) = \texttt{LL} \wedge \mu'(x) \in \mathcal{N}$ is a nonce$\}$.

Similarly α' is such that $\forall x \in \text{dom}(\alpha').\forall n.\ \alpha(x) = n \Leftrightarrow \alpha'(x\delta_i^1) = n\delta_i^1$, *i.e.* $\delta_i^1\alpha'\delta_i^1 = \alpha$.

By assumption, $[\,C\,]_2^n$ is not empty, hence there exists $(c'', \Gamma'') \in [\,C\,]_2^n$. Moreover, as noted previously, $(c', \Gamma'^i) \in [\,C'\,]_1^n$. Thus, $(c_i^{i,i} \cup c'' \cup c', \Gamma_i^{i,i} \cup \Gamma'' \cup \Gamma'^i) \in [\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n$.

Hence, by assumption, $\texttt{check\_const}(\{(c_i^{i,i} \cup c'' \cup c', \Gamma_i^{i,i} \cup \Gamma'' \cup \Gamma'^i)\})$ succeeds. If we have $\overline{c}'' = fst(\texttt{step1}_{\Gamma''}(c''))$, then $\overline{c}_i^{i,i} \cup \overline{c}'' \cup \overline{c'}^i = fst(\texttt{step1}_{\Gamma_i^{i,i} \cup \Gamma'' \cup \Gamma'^i}(c_i^{i,i} \cup c'' \cup c'))$.

Thus, $\texttt{step3}_{\overline{\Gamma}_i^{i,i} \cup \overline{\Gamma}'' \cup \overline{\Gamma'}^i}(\overline{c}_i^{i,i} \cup \overline{c}'' \cup \overline{c'}^i)$ holds, and since $\{M_1' \sim N_1', M_2' \sim N_2'\} \subseteq \overline{c}_i^{i,i} \cup \overline{c}'' \cup \overline{c'}^i$, we know that $N_1'\alpha' = N_2'\alpha'$.

Thus $N_1'\alpha'\delta_i^1 = N_2'\alpha'\delta_i^1$, *i.e.*, since $\delta_i^1\alpha'\delta_i^1 = \alpha$, $N_1\alpha = N_2\alpha$.

Finally, only the case where $M_1 \sim N_1 \in \overline{c}'$ and $M_2 \sim N_2 \in \overline{c}'$ remains. By assumption, $[\,C\,]_1^n$ and $[\,C\,]_2^n$ are not empty, hence there exist $(c'', \Gamma'') \in [\,C\,]_1^n$ and $(c''', \Gamma''') \in [\,C\,]_2^n$. Thus, $(c'' \cup c''' \cup c', \Gamma'' \cup \Gamma''' \cup \Gamma') \in [\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n$.

Hence, by assumption, $\texttt{check\_const}(\{(c'' \cup c''' \cup c', \Gamma'' \cup \Gamma''' \cup \Gamma')\})$ succeeds. If $\overline{c}'' = fst(\texttt{step1}_{\Gamma''}(c''))$ and $\overline{c}''' = fst(\texttt{step1}_{\Gamma'''}(c'''))$, then $\overline{c}'' \cup \overline{c}''' \cup \overline{c}' = fst(\texttt{step1}_{\Gamma'' \cup \Gamma''' \cup \Gamma'}(c'' \cup c''' \cup c'))$.

Thus, $\texttt{step3}_{\overline{\Gamma}'' \cup \overline{\Gamma}''' \cup \overline{\Gamma}'}(\overline{c}'' \cup \overline{c}''' \cup \overline{c}')$ holds, and since $\{M_1 \sim N_1, M_2 \sim N_2\} \subseteq \overline{c}'' \cup \overline{c}''' \cup \overline{c}'$, we know that $N_1\alpha = N_2\alpha$.

Therefore the claim holds in this case, which concludes the proof that $\texttt{step3}_{\overline{\Gamma}}(\overline{c})$ holds.

Therefore, for every $(c, \Gamma) \in (\cup_{\times 1 \le i \le n} [\,C\,]_i^n) \cup_\times [\,C'\,]_1^n$, $\texttt{check\_const}(\{(c, \Gamma)\})$ succeeds, which proves the claim. □

Lemma A.42. *For all* (c, Γ) *such that* $\text{vars}(c) \subseteq \text{dom}(\Gamma)$ *which only contains variables indexed by 1 or 2, and all names and keys in* c *have finite types, if* $\texttt{check\_const}(\{(c, \Gamma)\})$ *succeeds, then for all* $\Gamma'' \in \text{branches}(\Gamma')$*, where*

$$\Gamma' = \Gamma[\bigvee_{1 \le i \le n} [\![\tau_{m_i}^{l,1}\,;\, \tau_{p_i}^{l',1}]\!] \,/\, [\![\tau_m^{l,\infty}\,;\, \tau_p^{l',\infty}]\!]]_{m,p \in \mathcal{N}}[\text{seskey}^{l,1}(T) \,/\, \text{seskey}^{l,\infty}(T)],$$

$\texttt{check\_const}(\{(c, \Gamma'')\})$ *succeeds.*

Proof. Let $n \in \mathbb{N}$.

Let (c, Γ) be as defined in the statement of the lemma.

Let us assume that $\texttt{check\_const}(\{(c, \Gamma)\})$ succeeds. Let

$$\Gamma' = \Gamma[\bigvee_{1 \le i \le n} [\![\tau_{m_i}^{l,1}\,;\, \tau_{p_i}^{l',1}]\!] \,/\, [\![\tau_m^{l,\infty}\,;\, \tau_p^{l',\infty}]\!]]_{m,p \in \mathcal{N}}[\text{seskey}^{l,1}(T) \,/\, \text{seskey}^{l,\infty}(T)],$$

and let $\Gamma'' \in \text{branches}(\Gamma')$.

The procedure $\texttt{check\_const}(\{(c, \Gamma'')\})$ is as follows:

1. We compute $(\overline{c}, \overline{\Gamma}'') = \texttt{step1}_{\Gamma''}(c)$. Following the notations in the procedure, we denote

$$F = \{x \in \mathrm{dom}(\Gamma'') \mid \exists m, p, l, l'.\ \Gamma''(x) = [\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!]\}.$$

   Let $F' = \{x \in \mathrm{dom}(\Gamma) \mid \exists m, p, l, l'.\ \Gamma(x) = [\![\tau_m^{l,1}\,;\,\tau_p^{l',1}]\!]\}$.

   Let also $F'' = \{x \in \mathrm{dom}(\Gamma) \mid \exists m, p, l, l'.\ \Gamma(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]\}$.

   It is easily seen from the definition of Γ' that $F = F' \uplus F''$.

   By definition of $\texttt{step1}_{\Gamma''}(c)$, $\overline{\Gamma}''$ contains $\Gamma''|_{\mathrm{dom}(\Gamma'')\setminus F}$.

   Let $(\overline{c}', \overline{\Gamma}) = \texttt{step1}_{\Gamma}(c)$.

   It is clear from the definitions of Γ' and Γ'' that for all $x \in F''$, there exists $i \in [\![1, n]\!]$ and m, p, l, l' such that $\Gamma(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$ and $\Gamma''(x) = [\![\tau_{m_i}^{l,1}\,;\,\tau_{p_i}^{l',1}]\!]$. Let σ_ℓ and σ_r be the substitutions defined by

$$\mathrm{dom}(\sigma_\ell) = \mathrm{dom}(\sigma_r) = F''$$

   and

$$\forall x \in F''. \forall m, p \in \mathcal{N}.\ \forall l, l'.\ \forall i \in [\![1, n]\!].\ \Gamma''(x) = [\![\tau_{m_i}^{l,1}\,;\,\tau_{p_i}^{l',1}]\!] \Rightarrow (\sigma_\ell(x) = m_i\ \wedge\ \sigma_r(x) = p_i).$$

   It is clear from the definition of $\overline{c}$ and $\overline{c}'$ that $\overline{c} = [\![\overline{c}']\!]_{\sigma_\ell, \sigma_r}$.

2. We check that $\texttt{step2}_{\overline{\Gamma}''}(\overline{c})$ holds.

   Let $u \sim v \in \overline{c}$. Since $\overline{c} = [\![\overline{c}']\!]_{\sigma_\ell, \sigma_r}$, there exists $u' \sim v' \in \overline{c}'$ such that $u = u'\sigma_\ell$ and $v = v'\sigma_r$.

   Since $\texttt{check\_const}(\{(c, \Gamma)\}) = \texttt{true}$, we know that u' and v' have the required form. Note that by definition of $\overline{\Gamma}''$, the keys which are secret in $\overline{\Gamma}''$, *i.e.* the keys $k \in \mathcal{K}$ such that there exist k', T such that $\overline{\Gamma}''(k, k') <: \mathrm{key}^{\mathtt{HH}}(T)$ or $\overline{\Gamma}''(k', k) <: \mathrm{key}^{\mathtt{HH}}(T)$, are exactly the keys which are secret in $\overline{\Gamma}$. Similarly, for all variable x, there exists T such that $\overline{\Gamma}(x) <: \mathrm{key}^{\mathtt{HH}}(T)$ if and only if there exists T such that $\overline{\Gamma}''(x) <: \mathrm{key}^{\mathtt{HH}}(T)$; and there exist m, n, a such that $\overline{\Gamma}(x) = [\![\tau_m^{\mathtt{HH},a}\,;\,\tau_n^{\mathtt{HH},a}]\!]$ if and only if there exist m, n, a such that $\overline{\Gamma}''(x) = [\![\tau_m^{\mathtt{HH},a}\,;\,\tau_n^{\mathtt{HH},a}]\!]$.

   It clearly follows, by examining all cases for u' and v', that $u'\sigma_\ell$ and $v'\sigma_r$, *i.e.* u and v, also have the required form.

   Therefore, $\texttt{step2}_{\overline{\Gamma}''}(\overline{c})$ holds.

3. Finally, we check the condition $\texttt{step3}_{\overline{\Gamma}''}(\overline{c})$.

   Let $M_1 \sim N_1 \in \overline{c}$ and $M_2 \sim N_2 \in \overline{c}$. Since $\overline{c} = [\![\overline{c}']\!]_{\sigma_\ell, \sigma_r}$, there exist $M_1' \sim N_1' \in \overline{c}'$ and $M_2' \sim N_2' \in \overline{c}'$ such that $M_1 = M_1'\sigma_\ell$, $N_1 = N_1'\sigma_r$, $M_2 = M_2'\sigma_\ell$, and $N_2 = N_2'\sigma_r$.

   Let us prove the first direction of the equivalence, *i.e.* the case where M_1, M_2 are unifiable. The proof for the case where N_1, N_2 are unifiable is similar.

   If M_1, M_2 are unifiable, let μ be their most general unifier. We have $M_1\mu = M_2\mu$, *i.e.* $(M_1'\sigma_\ell)\mu = (M_2'\sigma_\ell)\mu$.

   Let τ denote the substitution $\sigma_\ell\mu$. Since $M_1'\tau = M_2'\tau$, M_1' and M_2' are unifiable. Let μ' be their most general unifier. There exists θ such that $\tau = \mu'\theta$.

   Let also α be the restriction of μ to $\{x \in \mathrm{vars}(M_1) \cup \mathrm{vars}(M_2) \mid \overline{\Gamma}''(x) = \mathtt{LL}\ \wedge\ \mu(x) \in \mathcal{N}\}$.

   Note that $\overline{\Gamma}''(x) = \mathtt{LL} \Leftrightarrow \overline{\Gamma}(x) = \mathtt{LL}$.

We have to prove that $N_1\alpha = N_2\alpha$.

Let $x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2')$ such that there exist m, p, l, l' such that $\overline{\Gamma}(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$, *i.e.* $x \in F''$. By definition of σ_ℓ (point 1), there exists $i \in [\![1,n]\!]$ such that $x\sigma_\ell = m_i$ (and $x\sigma_r = p_i$). Hence, we have

$$(x\mu')\theta = x\tau = x\sigma_\ell\mu = (x\sigma_\ell)\mu = m_i\mu = m_i.$$

Thus, $x\mu'$ can only be either a variable y such that $y\theta = m_i$, or the nonce m_i.

Therefore, μ' satisfies the conditions on the most general unifier expressed in $\mathtt{step3}_{\overline{\Gamma}}(\overline{c}')$.

Let $x \in \mathrm{vars}(M_1) \cup \mathrm{vars}(M_2)$ such that $\overline{\Gamma}''(x) = \mathtt{LL}$ and $\mu(x) \in \mathcal{N}$. We have $(x\mu')\theta = x\tau = x\sigma_\ell\mu = (x\sigma_\ell)\mu = x\mu = \mu(x) \in \mathcal{N}$. Thus, $x\mu'$ can only be either a variable y (such that $y\theta = \mu(x)$), or the nonce $\mu(x)$.

Conversely, let $x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2')$ such that $\overline{\Gamma}(x) = \mathtt{LL}$ and $\mu'(x) \in \mathcal{N}$. We have $x\mu = (x\sigma_\ell)\mu = x\tau = (x\mu')\theta = \mu'(x)$.

Let then θ' be the substitution with domain $\{x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2') \mid \exists m,p,l,l'.\exists i \in [\![1,n]\!].\ \overline{\Gamma}(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!] \wedge \mu'(x) = m_i\}$ such that $\forall x \in \mathrm{dom}(\theta').\ \theta'(x) = p_i$ if $\mu'(x) = m_i$ and $\overline{\Gamma}(x) = [\![\tau_m^{l,\infty}\,;\,\tau_p^{l',\infty}]\!]$.

Let also α' be the restriction of μ' to $\{x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2') \mid \overline{\Gamma}(x) = \mathtt{LL} \wedge \mu'(x) \in \mathcal{N}\}$.

Since $\mathtt{check\_const}(\{(c,\Gamma)\}) = \mathtt{true}$, we know that $\mathtt{step3}_{\overline{\Gamma}}(\overline{c}')$ holds. Since $M_1' \sim N_1' \in \overline{c}'$, and $M_2' \sim N_2' \in \overline{c}'$, this implies that $N_1'\alpha'\theta' = N_2'\alpha'\theta'$.

As we have just shown, for all $x \in \mathrm{dom}(\theta')$, there exists $i \in [\![1,n]\!]$ such that $x\sigma_\ell = m_i$ and $x\sigma_r = p_i$, and $\mu'(x)$ is either m_i or a variable. By definition of $\mathrm{dom}(\theta')$, only the case where $\mu'(x) = m_i$ is actually possible, and we have $\theta'(x) = p_i$.

Thus, $\forall x \in \mathrm{dom}(\theta').\ \sigma_r(x) = \theta'(x)$.

It then is clear from the definitions of the domains of θ' and σ_r that there exists τ' such that $\sigma_r = \theta'\tau'$.

Thus, since we have shown that $N_1'\alpha'\theta' = N_2'\alpha'\theta'$, we have $(N_1'\alpha'\theta')\tau' = (N_2'\alpha'\theta')\tau'$, that is to say $N_1'\alpha'\sigma_r = N_2'\alpha'\sigma_r$, *i.e.*, since α' and σ_r have disjoint domains, and are both ground, $N_1\alpha' = N_2\alpha'$.

Moreover, we have shown that for all $x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2')$ such that $\overline{\Gamma}(x) = \mathtt{LL}$ and $\mu'(x) \in \mathcal{N}$, $\mu(x) = \mu'(x)$. That is to say that for all $x \in \mathrm{dom}(\alpha')$, $\mu(x) = \alpha'(x)$.

In addition, it is clear from the definition of σ_ℓ that

$$\{x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2') \mid \overline{\Gamma}(x) = \mathtt{LL}\} = \{x \in \mathrm{vars}(M_1) \cup \mathrm{vars}(M_2) \mid \overline{\Gamma}''(x) = \mathtt{LL}\}.$$

Hence

$$\begin{aligned}
\mathrm{dom}(\alpha) &= \{x \in \mathrm{vars}(M_1) \cup \mathrm{vars}(M_2) \mid \overline{\Gamma}''(x) = \mathtt{LL} \wedge \mu(x) \in \mathcal{N}\} \\
&= \{x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2') \mid \overline{\Gamma}(x) = \mathtt{LL} \wedge \mu(x) \in \mathcal{N}\} \\
&\supseteq \{x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2') \mid \overline{\Gamma}(x) = \mathtt{LL} \wedge \mu(x) \in \mathcal{N} \wedge \mu'(x) \in \mathcal{N}\} \\
&= \{x \in \mathrm{vars}(M_1') \cup \mathrm{vars}(M_2') \mid \overline{\Gamma}(x) = \mathtt{LL} \wedge \mu'(x) \in \mathcal{N}\} \\
&= \mathrm{dom}(\alpha').
\end{aligned}$$

Therefore, $\forall x \in \mathrm{dom}(\alpha').\ x \in \mathrm{dom}(\alpha) \ \wedge\ \alpha'(x) = \alpha(x)$. Thus there exists α'' such that $\alpha = \alpha'\alpha''$.

Since we already have $N_1\alpha' = N_2\alpha'$, this implies that $N_1\alpha = N_2\alpha$, which concludes the proof that $\mathtt{step3}_{\overline{\Gamma}''}(\overline{c})$ holds. Hence, $\mathtt{check\_const}(\{(c, \Gamma'')\}) = \mathtt{true}$.

□

Theorem A.8. *Let C, and C' be two constraint sets without any common variable*

$$\mathtt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \mathtt{true} \ \Rightarrow \\ \forall n.\ \mathtt{check\_const}([\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n) = \mathtt{true}.$$

Proof. Assume $\mathtt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \mathtt{true}$. Let $n > 0$.

Let us show that $\mathtt{check\_const}([\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n) = \mathtt{true}$.

By assumption, we know that $\mathtt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \mathtt{true}$.

That is to say, for each $(c_1, \Gamma_1) \in C$, $(c_2, \Gamma_2) \in C$, $(c_3, \Gamma_3) \in C'$, if $c' = [\,c_1\,]_1^{\Gamma_1} \cup [\,c_2\,]_2^{\Gamma_2} \cup [\,c_3\,]_1^{\Gamma_3}$, and $\Gamma' = [\,\Gamma_1\,]_1 \cup [\,\Gamma_2\,]_2 \cup [\,\Gamma_3\,]_1$, $\mathtt{check\_const}(\{(c', \Gamma')\}) = \mathtt{true}$.

Thus, by Lemma A.42, for all $(c_1, \Gamma_1) \in C$, $(c_2, \Gamma_2) \in C$, $(c_3, \Gamma_3) \in C'$, if $c' = [\,c_1\,]_1^{\Gamma_1} \cup [\,c_2\,]_2^{\Gamma_2} \cup [\,c_3\,]_1^{\Gamma_3}$, and $\Gamma' = [\,\Gamma_1\,]_1^n \cup [\,\Gamma_2\,]_2^n \cup [\,\Gamma_3\,]_1^n$, $\mathtt{check\_const}(\{(c', \Gamma')\}) = \mathtt{true}$.

That is to say, $\mathtt{check\_const}([\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n) = \mathtt{true}$, which concludes the proof. □

Theorem A.9. *Let C, and C' be two constraint sets without any common variable. Then*

$$\mathtt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \mathtt{true} \ \Rightarrow \forall n.\ [\,C'\,]_1^n \cup_\times (\cup_{\times 1 \le i \le n} [\,C\,]_i^n) \textit{ is consistent.}$$

Proof. Assume $\mathtt{check\_const}([\,C\,]_1 \cup_\times [\,C\,]_2 \cup_\times [\,C'\,]_1) = \mathtt{true}$, and let $n \geq 1$. By Theorem A.8, we have

$$\mathtt{check\_const}([\,C\,]_1^n \cup_\times [\,C\,]_2^n \cup_\times [\,C'\,]_1^n) = \mathtt{true}.$$

Then by Theorem A.7:

$$\mathtt{check\_const}(\cup_{\times 1 \le i \le n} [\,C\,]_i^n) \cup_\times [\,C'\,]_1^n) = \mathtt{true}.$$

Therefore, by Theorem A.6, $(\cup_{\times 1 \le i \le n} [\,C\,]_i^n) \cup_\times [\,C'\,]_1^n$ is consistent, which concludes the proof. □

Appendix B

Proofs of the case study (Chapter 6)

In this appendix, we provide the detailed proofs for the case study in Chapter 6, detailing all assumptions we made and defining the relevant RECOVER algorithms.

B.1 Perfect ideal functionality

We show here the result mentioned in Section 6.6.2, *i.e.* that, assuming all voters verify their vote, and that the counting function ρ is stable, the perfect ideal functionality $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$ is implied by the functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$.

Recall that a result function ρ is *stable* if changing the order of votes does not change the result, as long as for each voter, her votes remain in the same order, *i.e.*

$$\forall L, L'. \quad (\forall id.\, [(id', v) \in L | id' = id] = [(id', v) \in L' | id' = id]) \;\Rightarrow\; \rho(L) = \rho(L').$$

We prove the following theorem.

Theorem B.10. *Assume that* $\mathsf{H} = \mathsf{H}_{\mathsf{check}}$*, and that* ρ *is a stable counting function. For any simulator* $\mathcal{S}$*, there exists a simulator* $\mathcal{S}'$ *such that for any well-behaved environment* $\mathcal{E}$*, the outputs of* $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$ *and* $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$ *are indistinguishable.*

Proof. Let $\mathcal{S}$ be a simulator, intended to be executed with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$. We construct a simulator $\mathcal{S}'$, intended to be executed with $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$. $\mathcal{S}'$ will run $\mathcal{S}$ internally, and answers queries from $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$ as follows.

- On `setup` from $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, starting the setup phase:
  $\mathcal{S}'$ sends `setup` to $\mathcal{S}$ (in the name of $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$).
- On `voting` from $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, starting the voting phase:
  $\mathcal{S}'$ sends `voting` to $\mathcal{S}$.
- On `ack`(id) from $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$ during the voting phase:
  $\mathcal{S}'$ forwards this message to $\mathcal{S}$, and appends id to a list LL (initially empty). Let us call L the list of (identities,votes) kept by the functionality $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$ in the execution $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$. It is clear that LL is the list of identities in L, *i.e.* $LL = [id|(id, v) \in L]$. In addition, by definition of $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ and $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, L is also equal to the list kept by $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ in the (simulated) execution $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$.

- On `voting done` from $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$:
  $\mathcal{S}'$ forwards this message to $\mathcal{S}$.
- On `modif?` from $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$:
  - $\mathcal{S}'$ forwards `modif?` to $\mathcal{S}$, and waits for $\mathcal{S}$ to try to send a modification function to $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$. When $\mathcal{S}$ sends `modif`(f), $\mathcal{S}'$ retrieves the function f, and has to produce a function to send to $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$.
  - $\mathcal{S}'$ starts by checking whether $P^{\mathsf{del}}(L, f) = \top$. $\mathcal{S}'$ does not have access to the list L, but it appears by carefully examining the definition of P^{del} that L itself is not needed to compute $P^{\mathsf{del}}(L, f)$: only the list of the identities in L is used. $\mathcal{S}'$ knows this list of identities: as previously noted, it is equal to LL.
  - If $P^{\mathsf{del}}(L, f) = \bot$, $\mathcal{S}'$ sends `modif`$(f_\emptyset)$ to $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, where $f_\emptyset : \emptyset \to \emptyset$ is the empty function.
  - Otherwise, $\mathcal{S}'$ computes the list LL_{cast} of the votes cast in f:

    $$LL_{\mathsf{cast}} = [f(i), i = 1 \dots |\mathrm{dom}(f)| \mid \exists (id, v).\ f(i) = (id, v).$$

    $\mathcal{S}'$ then computes the list $L' = [1, \dots, |LL|] \parallel LL_{\mathsf{cast}}$. Finally $\mathcal{S}'$ sends `modif`$(\lambda i.\ L'[i])$ to $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$.
- On `result`(r) from $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, for some r (potentially `no tally`), in the tallying phase:
  - If $P^{\mathsf{del}}(L, f)$ was $\bot$ in the previous step: $\mathcal{S}'$ sends `result`(`no tally`) to $\mathcal{S}$, and regardless of its answer, sends `res-block` to $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$.
  - Otherwise, $\mathcal{S}'$ sends `result`(r) to $\mathcal{S}$, and waits for $\mathcal{S}$'s answer. $\mathcal{S}$ either answers `res-ok` or `res-block`. $\mathcal{S}$ forwards this message to $\mathcal{F}_{\mathsf{v}}^{P}(\rho)$.
- In addition, during the whole execution, $\mathcal{S}$ forwards between $\mathcal{E}$ and $\mathcal{S}$ any message they wish to exchange.

Let us now prove that the outputs of $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$ and $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$ are indistinguishable. We do this by showing that both $\mathcal{E}$ and the simulated $\mathcal{S}$ in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$ have the same view they would have in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$.

It is clear that up to right before $\mathcal{S}'$ sends the `result` command to $\mathcal{S}$, these views are indeed the same. The subtle points is that remain to be proved are then that

- $\mathcal{S}$ simulated by $\mathcal{S}'$ receives exactly the same result it would receive in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$;
- and $\mathcal{E}$ in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$ receives the same result (or `no tally`) from $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$ as it would receive from $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ in the execution with $\mathcal{S}$ $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$.

Let us first deal with the easy case where the test $P^{\mathsf{del}}(L, f)$ performed by $\mathcal{S}'$ fails. In that case, in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ would reject f, meaning that $\mathcal{S}$ would receive `result`(`no tally`), and $\mathcal{E}$ would receive `no tally`. These are indeed the results they see in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$.

The interesting case is when $P^{\mathsf{del}}(L, f) = \top$. Then in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ would accept f, and compute $r = \rho(\overline{f}(L))$. $\mathcal{S}$ would receive `result`(r), and $\mathcal{E}$ would receive r or

`no tally` depending on $\mathcal{S}$'s decision. In $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$, $\mathcal{S}'$ computes $f' = \lambda i.\ L'[i]$, where $L' = [1, \ldots, |LL|] \parallel LL_{\mathsf{cast}}$ and LL_{cast} are the votes cast in f. This f' is clearly accepted by $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, *i.e.* $P^{\emptyset}(L, f') = \top$. Indeed

- f' keeps the votes of all honest voters in the same order:

$$[f'(j), j = 1 \ldots |\mathrm{dom}(f')| \mid f'(j) \in \mathbb{N}] = [1, \ldots, |LL|] = [1, \ldots, |L|]$$

- and no honest votes are modified by f': $\forall i, id, v$, if $f'(i) = (id, v)$ then $(id, v) \in \mathrm{Im}(f)$, which implies, since $P^{\mathsf{del}}(L, f) = \top$, that $id \in \mathsf{D}$.

Thus $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$ accepts f', and returns to $\mathcal{S}'$ $r' = \rho(\overline{f'}(L)) = \rho(L \parallel LL_{\mathsf{cast}})$. In addition, since $P^{\mathsf{del}}(L, f) = \top$, f keeps the votes of each separate voter $id \in \mathsf{H}$ (as $\mathsf{H} = \mathsf{H}_{\mathsf{check}}$) in the same order. Thus it is clear that $L \parallel LL_{\mathsf{cast}}$ and $\overline{f}(L)$ feature the votes of each voter in the same order. By assumption ρ is stable, and therefore $r = r'$. Thus, $\mathcal{S}$ and $\mathcal{E}$ indeed receive the same result as in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$, which concludes the proof. □

We can similarly show the following theorem, when there is no revote.

Theorem B.11. *Assume that* $\mathsf{H} = \mathsf{H}_{\mathsf{check}}$*, and that* ρ *is a stable counting function. For any simulator* $\mathcal{S}$*, there exists a simulator* $\mathcal{S}'$ *such that for any well-behaved environment* $\mathcal{E}$ *that does not make voters revote, the outputs of* $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho))$ *and* $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$ *are indistinguishable.*

Proof. The construction of the simulator $\mathcal{S}'$ is exactly the same as in the proof of the previous theorem, except that it checks whether $P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(L, f) = \top$ instead of $P^{\mathsf{del}}(L, f) = \top$. As before, we show that both $\mathcal{E}$ and the simulated $\mathcal{S}$ in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$ have the same view they would have in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho))$.

For the same reason as in the previous proof, it is clear that these views are the same, up to right before $\mathcal{S}'$ sends the `result` command to $\mathcal{S}$. The subtle point, as before, is to prove that $\mathcal{E}$ and the simulated $\mathcal{S}$ in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$ both receive the same result they would obtain in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho))$. Here, too, the case where $P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(L, f) = \bot$ is trivial, as both receive `no tally`.

The interesting case is when $P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(L, f) = \top$. Then in the ideal execution $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho))$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho)$ would accept f, and compute $r = \rho(\overline{f}(L))$. $\mathcal{S}$ would receive $\mathtt{result}(r)$, and $\mathcal{E}$ would receive r or `no tally` depending on $\mathcal{S}$'s decision. In $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$, $\mathcal{S}'$ computes $f' = \lambda i.\ L'[i]$, where $L' = [1, \ldots, |LL|] \parallel LL_{\mathsf{cast}}$ and LL_{cast} are the votes cast in f. This f' is clearly accepted by $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$, *i.e.* $P^{\emptyset}(L, f') = \top$. Indeed

- f' keeps the votes of all honest voters in the same order:

$$[f'(j), j = 1 \ldots |\mathrm{dom}(f')| \mid f'(j) \in \mathbb{N}] = [1, \ldots, |LL|] = [1, \ldots, |L|]$$

- and no honest votes are modified by f': $\forall i, id, v$, if $f'(i) = (id, v)$ then $(id, v) \in \mathrm{Im}(f)$, which implies, since $P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(L, f) = \top$, that $id \in \mathsf{D} \cup \mathsf{H}_{\overline{\mathsf{check}}} = \mathsf{D}$.

Thus $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$ accepts f', and returns to $\mathcal{S}'$ $r' = \rho(\overline{f'}(L)) = \rho(L \parallel LL_{\mathsf{cast}})$. In addition, since $P^{\mathsf{del,reorder,change}}(L, f) = \top$, f keeps the (unique, as $\mathcal{E}$ does not make voters revote) vote of all voters $id \in \mathsf{H}$ (as $\mathsf{H} = \mathsf{H}_{\mathsf{check}}$). Thus it is clear that $L \parallel LL_{\mathsf{cast}}$ and $\overline{f}(L)$ contain the same votes, and only differ by the order of the honest votes. Thus they feature the votes of each voter in the same order. By assumption ρ is stable, and therefore $r = r'$. Thus, as in the previous proof, $\mathcal{S}$ and $\mathcal{E}$ indeed receive the same result as in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$, which concludes the proof. □

From the previous two theorems, we get that:

Theorem B.12. *Let $\mathcal{V}$ be a voting scheme. Assume that $\mathsf{H} = \mathsf{H}_{\mathsf{check}}$, and that ρ is a stable counting function.*

- *If $\mathcal{V}$ securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$, then $\mathcal{V}$ securely implements $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$.*
- *If we only consider environments where voters do not revote, if $\mathcal{V}$ securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$, then $\mathcal{V}$ securely implements $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$.*

Proof. The first point directly follows from Theorem B.10. Indeed, if $\mathcal{V}$ securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$, then for any adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for any well-behaved $\mathcal{E}$ the outputs of $\mathsf{realexec}(\mathcal{E}||\mathcal{A}||\mathcal{V})$ and $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$ are indistinguishable. By Theorem B.10 there exists $\mathcal{S}'$ such that for any well-behaved $\mathcal{E}$ the outputs of $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}'||\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho))$ and $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$ are indistinguishable. This $\mathcal{S}'$ proves that $\mathcal{V}$ securely implements $\mathcal{F}_{\mathsf{v}}^{\emptyset}(\rho)$.

The second point similarly follows from Theorem B.11. □

B.2 Case study: Civitas and Belenios without revote

We show here that, under reasonable assumptions, Civitas and Belenios without revote are mb-BPRIV *w.r.t.* $\textsc{Recover}^{\mathsf{del,reorder}}$ (defined below), and that this implies they realise $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$. Equivalently, as there is no revote, they realise $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$. Consequently they also realise the weaker $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$.

We write the assumptions and proofs in a generic way, so that they apply to both Civitas and Belenios.

B.2.1 Notations: Civitas

$\mathsf{enc}(\cdot, \mathsf{pk})$ denotes ElGamal encryption under key pk.

- $\mathsf{Register}(id)$ generates a private credential c_{id}
- $\mathsf{Pub}(c) = \mathsf{enc}(c, \mathsf{pk})$ encrypts the credential c
- $\mathsf{Vote}(id, \mathsf{pk}, c, v) = (s, \mathsf{enc}(c, \mathsf{pk}), (\mathsf{enc}(v, \mathsf{pk}), \pi_1, \pi_2))$. The state s records the ballot. The public credential/pseudonym is $\mathsf{enc}(c, \mathsf{pk})$. π_1 is a zero-knowledge proof that v is a valid vote, and π_2 is a zero-knowledge proof that the agent generating the ballot knows both c and v.
- $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk})$ checks for each (p, b) in BB the zero-knowledge proofs in b, and ensures that p is a valid public credential, using $\mathsf{Pub}(\mathsf{U})$ and a PET. It finally checks, also with a PET, that BB does not already contain several ballots (p, b), (p', b') where p and p' encrypt the same credential.

- Tally only keeps the parts of the ballots containing the encrypted votes. These are then run through a mixnet, decrypted, and published.

- $\mathsf{Verify}(id, s, \mathsf{BB})$ checks whether the ballot recorded in s appears on BB.

B.2.2 Notations: Belenios

$\mathsf{signElGamal}(\cdot, \mathsf{pk}, k)$ denotes the combination of ElGamal encryption and signature, under the public encryption key pk and the private signing key k.

- $\mathsf{Register}(id)$ generates a pair (k_{id}, pk_{id}) of a secret signing key k_{id} and the associated verification key pk_{id}.

- $\mathsf{Pub}(k_{id}, pk_{id}) = pk_{id}$ is simply the public verification key.

- $\mathsf{Vote}(id, \mathsf{pk}, (k_{id}, pk_{id}), v) = (s, pk_{id}, (\mathsf{signElGamal}(v, \mathsf{pk}, k_{id}), \Pi))$. The state s records the ballot. The public credential/pseudonym is the public key pk_{id}. π is a zero-knowledge proof that v is a valid vote.

- $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk})$ checks for each (p, b) in BB the zero-knowledge proof in b, and ensures that p is a valid verification key, using $\mathsf{Pub}(\mathsf{U})$, and that b is indeed signed with the associated signing key. It finally checks that BB does not contain several ballots (p, b), (p, b') that use the same public key, *i.e.* such that $p = p'$.

- Tally can either run the ballots through a mixnet, decrypt them, and publish the decrypted votes; or homomorphically compute the sum of all ballots, before decrypting and publishing the result.

- $\mathsf{Verify}(id, s, \mathsf{BB})$ checks whether the ballot recorded in s appears on BB, signed by id's signing key.

B.2.3 Recovery

We consider the following recovery algorithm:

$\textsc{Recover}_{\mathsf{U}}^{\mathsf{del,reorder}}(\mathsf{BB}_1, \mathsf{BB})$

$L \leftarrow []$;
for $(p, b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $L \leftarrow L \parallel j$
    (in case several such j exist, pick the first one)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ **then**
    $L \leftarrow L \parallel (p, b)$
$L' \leftarrow [i | \mathsf{BB}_1[i] = (id, (p, b)) \ \wedge \ id \in \mathsf{H}_{\mathsf{check}} \ \wedge \ (p, b) \notin \mathsf{BB}]$
$L'' \leftarrow L \parallel L'$
return $(\lambda i.\ L''[i])$

B.2.4 Assumptions

We assume that the ballots (including the public credential) are non malleable, *i.e.* that for all adversary $\mathcal{A}$,

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|$$

is negligible in λ, where $\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{A},\mathcal{V}}$ is the following game.

$\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{A},\mathcal{V}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $\mathsf{U}[id] \leftarrow \mathsf{Register}(1^\lambda, id)$
$\beta' \leftarrow \mathcal{A}^{\mathcal{O}_\mathsf{c},\mathcal{O}_\mathsf{d}}(\mathsf{pk},\mathsf{U})$
output β'.

$\mathcal{O}_\mathsf{c}(id, c, v_0, v_1)$

$(p, b, state) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, c, v_\beta)$
$\mathsf{L} \leftarrow \mathsf{L} \parallel (p, b)$
return (p, b).

$\mathcal{O}_\mathsf{d}(\mathsf{cL})$

for all $(p, b) \in \mathsf{cL}$ **do**
  if $(p, b) \notin \mathsf{L}$ **then**
    $\mathsf{dL} \leftarrow \mathsf{dL} \parallel \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$
return dL.

$\mathcal{A}$ is allowed any number of calls to $\mathcal{O}_\mathsf{c}$, followed by one single call to $\mathcal{O}_\mathsf{d}$.

We also assume that the credentials within the ballots are non malleable, *i.e.* that for all adversary $\mathcal{A}$, $|\mathbb{P}(\mathsf{Exp}^{\mathsf{CNM}}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|$ is negligible in λ, where $\mathsf{Exp}^{\mathsf{CNM}}_{\mathcal{A},\mathcal{V}}$ is the following game.

$\mathsf{Exp}^{\mathsf{CNM}}_{\mathcal{A},\mathcal{V}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $c \leftarrow \mathsf{Register}(1^\lambda, id); \mathsf{U}[id] \leftarrow c; \mathsf{PU}[id] \leftarrow \mathsf{Pub}(c)$
for all $id \in \mathsf{D}$ **do** $\mathsf{CU}[id] \leftarrow \mathsf{U}[id]$
$\mathsf{BB} \leftarrow \mathcal{A}^{\mathcal{O}_\mathsf{c}}(\mathsf{pk}, \mathsf{CU}, \mathsf{PU})$
if $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) \ \wedge \ \exists (p, b) \in \mathsf{BB}.\ (p, b) \notin \mathsf{L}$
  $\wedge \ \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) = (id, *)$
  for some $id \in \mathsf{H}$
then return 1
else return 0.

$\mathcal{O}_\mathsf{c}(id, v)$

$(p, b, state) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$
$\mathsf{L} \leftarrow \mathsf{L} \parallel (p, b)$
return (p, b).

We assume that the voting scheme is strongly consistent.

We also assume a zero-knowledge property from the proof of correct tallying, expressed by the following game:

$\mathsf{Exp}^{\mathsf{ZK},\beta}_{\mathcal{A},\mathcal{V}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
$\mathsf{BB} \leftarrow \mathcal{A}(\mathsf{pk})$
$(r, \Pi_0) \leftarrow \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$
$\Pi_1 \leftarrow \mathsf{SimProof}(r, \mathsf{BB})$
$\beta' \leftarrow \mathcal{A}(r, \Pi_\beta)$
output β'.

Finally we assume that the proof of correct tallying and the rest of the protocol use different random oracles.

B.2.5 Belenios and Civitas without revote are mb-BPRIV

Theorem B.13. *Assume Belenios, with no revote allowed, is strongly consistent, has non-malleable ballots and credentials, that the proofs of correct tallying are zero-knowledge, and that different random oracles are used for these proofs and the rest of the protocol. Then Belenios, without revote, is mb-BPRIV* w.r.t. $\textsc{Recover}^{\mathsf{del},\mathsf{reorder}}$.

Under the same assumptions, Civitas without revote also is mb-BPRIV w.r.t. $\textsc{Recover}^{\mathsf{del},\mathsf{reorder}}$.

Proof. Consider the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$, which is identical to $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover}^{\mathsf{del},\mathsf{reorder}},\beta}_{\mathcal{A},\mathcal{V}}$, except that the adversary is not shown the tally of the board, but instead the result computed using ρ and the $\mathsf{extract}$ function, without any proof of correct tallying. That is, the $\mathcal{O}\mathsf{tally}$ oracle is replaced with $\mathcal{O}\mathsf{tally}'$, defined by:

$\mathcal{O}\mathsf{tally}'_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 0$	$\mathcal{O}\mathsf{tally}'_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 1$
return $\rho(\mathsf{extract}(\mathsf{sk},\mathsf{U},\mathsf{BB}))$	$\pi \leftarrow \textsc{Recover}^{\mathsf{del},\mathsf{reorder}}_{\mathsf{U}}(\mathsf{BB}_1,\mathsf{BB})$
	$\mathsf{BB}' \leftarrow \overline{\pi}(\mathsf{BB}_0)$
	return $\rho(\mathsf{extract}(\mathsf{sk},\mathsf{U},\mathsf{BB}'))$

We will also consider the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},\beta}_{\mathcal{A},\mathcal{V}}$, where the adversary is given the result of the election as computed by the Tally algorithm, but still no proof of correct tallying, *i.e.* this game is identical to $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$, except the oracle $\mathcal{O}\mathsf{tally}$ is replaced with $\mathcal{O}\mathsf{tally}''$:

$\mathcal{O}\mathsf{tally}''_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 0$	$\mathcal{O}\mathsf{tally}''_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 1$
$(r,\Pi) \leftarrow \mathsf{Tally}(\mathsf{BB},\mathsf{sk})$	$\pi \leftarrow \textsc{Recover}^{\mathsf{del},\mathsf{reorder}}_{\mathsf{U}}(\mathsf{BB}_1,\mathsf{BB})$
return r	$\mathsf{BB}' \leftarrow \overline{\pi}(\mathsf{BB}_0)$
	$(r,\Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}',\mathsf{sk})$
	return r

Finally we consider a third variant, $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},\beta}_{\mathcal{A},\mathcal{V}}$, where the adversary is shown the result of the election, and the proof of correct tallying for the board provided by $\mathcal{A}$ is simulated on both sides, *i.e.* where the $\mathcal{O}\mathsf{tally}$ oracle is replaced with the following $\mathcal{O}\mathsf{tally}'''$:

$\mathcal{O}\mathsf{tally}'''_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 0$	$\mathcal{O}\mathsf{tally}'''_{\mathsf{BB},\mathsf{BB}_0,\mathsf{BB}_1}()$ for $\beta = 1$
$(r,\Pi) \leftarrow \mathsf{Tally}(\mathsf{BB},\mathsf{sk})$	$\pi \leftarrow \textsc{Recover}^{\mathsf{del},\mathsf{reorder}}_{\mathsf{U}}(\mathsf{BB}_1,\mathsf{BB})$
$\Pi' \leftarrow \mathsf{SimProof}(\mathsf{BB},r)$	$\mathsf{BB}' \leftarrow \overline{\pi}(\mathsf{BB}_0)$
return (r,Π')	$(r,\Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}',\mathsf{sk})$
	$\Pi' \leftarrow \mathsf{SimProof}(\mathsf{BB},r)$
	return (r,Π')

The structure of the proof is as follows. We first show that if no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{NM}}$ or $\mathsf{Exp}^{\mathsf{CNM}}$, then no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$. Using the strong consistency assumption, we then show that this implies no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. We then show, using the assumption that the random oracles used for the proof of correct tallying and the remainder of the protocol are different, that this implies no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. From there, using the zero-knowledge assumption on the proof of correct tallying, we show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$.

$\mathsf{Exp}^{\mathsf{NM}} \wedge \mathsf{Exp}^{\mathsf{CNM}} \Rightarrow \mathsf{Exp}^{\textbf{mb-BPRIV}1}$.

Let $\mathcal{A}$ be an attacker that wins the $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},}$ game.

We construct an attacker $\mathcal{B}$ against $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{NM},\beta}$. $\mathcal{B}$ is given access to pk and U. It runs $\mathcal{A}$ internally, simulating the oracle calls as follows. When $\mathcal{A}$ calls $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$, for some $id \in \mathsf{H}$, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{c}}(id, \mathsf{U}[id], v_0, v_1)$ and obtains some (p, b). $\mathcal{B}$ records $(id, (p, b))$ in a list $\mathsf{BB}_{\mathsf{init}}$, and stores (id, v_0, v_1, p, b) in a list V. $\mathcal{B}$ then returns (p, b) to $\mathcal{A}$. At some point, $\mathcal{A}$ returns to $\mathcal{B}$ some board BB.

Note that

- By construction, during the execution, the list $[(p, b)|(id, (p, b)) \in \mathsf{BB}_{\mathsf{init}}]$ of the ballots in $\mathsf{BB}_{\mathsf{init}}$ (kept by $\mathcal{B}$) is always equal to the list L in the game $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{NM},\beta}$ played by $\mathcal{B}$.
- It is also clear by examining the game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{mb\text{-}BPRIV1},\beta}$ that, up to this point, $\mathcal{A}$ has been accurately simulated, and has the same view it would have in game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{mb\text{-}BPRIV1},\beta}$.
- By construction of the $\mathcal{O}\mathsf{voteLR}$ oracle, the list $\mathsf{BB}_{\mathsf{init}}$ is also equal to the list BB_β in (the corresponding execution of) the game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{mb\text{-}BPRIV1},\beta}$ for $\mathcal{A}$.
- We can also note that $\mathsf{BB}_{\mathsf{init}}$ contains no duplicate ballots (except with a negligible probability $p_1^\beta(\lambda)$). Indeed, $\mathsf{BB}_{\mathsf{init}}$ containing duplicate ballots would imply that two oracle calls $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$ and $\mathsf{Vote}(\mathsf{pk}, id', \mathsf{U}[id'], v')$ produced the same ballot. If this happened with non-negligible probability, by submitting $\mathcal{O}_{\mathsf{c}}(id, \mathsf{U}[id], v, v'')$ followed by $\mathcal{O}_{\mathsf{c}}(id', \mathsf{U}[id'], v', v''')$ (for some $v'' \neq v'''$) to the encryption oracle, and checking whether the two resulting ballots are equal, an adversary would win the non-malleability game. The two ballots obtained on the right would indeed be different, except with negligible probability, by strong consistency (as they contain different votes).

Once $\mathcal{A}$ has returned BB, $\mathcal{B}$ checks (in V) that all voters in $\mathsf{H}_{\mathsf{check}}$ have voted, and halts if not. $\mathcal{B}$ also checks whether $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$. If not, $\mathcal{B}$ directly asks $\mathcal{A}$ to guess the bit β', and returns $\mathcal{A}$'s guess.

Following the structure of game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$, $\mathcal{B}$ then lets $\mathcal{A}$ perform the verifications for the honest voters. That is, when $\mathcal{A}$ calls $\mathcal{O}\mathsf{verify}_{\mathsf{BB}}(id)$, $\mathcal{B}$ retrieves the entry (id, v_0, v_1, p, b) for id in V, and checks that $(p, b) \in \mathsf{BB}$. Once $\mathcal{A}$ is done with the verification phase, $\mathcal{B}$ checks that each $id \in \mathsf{H}_{\mathsf{check}}$ has verified (and halts otherwise). If any of the verifications have failed, $\mathcal{B}$ directly asks $\mathcal{A}$ to guess the bit β', and returns $\mathcal{A}$'s guess.

If all verifications are successful, $\mathcal{B}$ will simulate the tallying phase to $\mathcal{A}$. To do this, $\mathcal{B}$ first computes $\pi = \textsc{Recover}_{\mathsf{U}}^{\mathsf{del},\mathsf{reorder}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB})$.

Note, at this point, that since all checks succeeded, π is simpler than in the general case. Indeed, this means that the algorithm $\textsc{Recover}_{\mathsf{U}}^{\mathsf{del},\mathsf{reorder}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB})$ did not need to add back any ballots from a honest voter who checked and whose ballot would be missing from BB. Hence, π is only defined on $[\![1, |\mathsf{BB}|]\!]$, and, except with negligible probability $p_2^\beta(\lambda)$, is such that for all i:

- $\pi(i) = j$ if j is the index of the first (and only) occurrence of $\mathsf{BB}[i]$ in $\mathsf{BB}_{\mathsf{init}}$,
- or $\pi(i) = \mathsf{BB}[i]$ if no such index exists. Indeed, in that case, we have $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ (where p is the public credential in $\mathsf{BB}[i]$), except with negligible probability: otherwise an adversary could win game $\mathsf{Exp}^{\mathsf{CNM}}$ by simulating $\mathcal{A}$ and returning BB.

To continue the simulation of $\mathcal{A}$, $\mathcal{B}$ must then provide $\mathcal{A}$ with the result of the election, to answer $\mathcal{A}$'s calls to $\mathcal{O}\mathsf{tally}'$. $\mathcal{B}$ asks for the decryption of the list of all (p,b) such that $\exists j.\ \pi(j) = (p,b)$. That is, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{d}}([(p,b)|\exists j.\ \pi(j) = (p,b)])$. This call to the decryption oracle is allowed: indeed, by definition of $\textsc{Recover}^{\mathsf{del},\mathsf{reorder}}$, a ballot such that $\pi(j) = (p,b)$ cannot occur in an element of $\mathsf{BB}_{\mathsf{init}}$, and hence is not in L.

$\mathcal{B}$ then constructs a list $\overline{\mathsf{BB}}$ containing the votes in $\overline{\pi}\mathsf{BB}_0$ in clear, BB_0 being the board of the left ballots, maintained by $\mathcal{O}\mathsf{voteLR}$ in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$. This is done by retrieving the vote from the list V for ballots coming from the encryption oracle, and using the decryption oracle for the others. Formally this list is obtained by, for all i:

- $\overline{\mathsf{BB}}[i] = (id, v_0)$ if $\mathsf{BB}[i] = (p,b)$ for some (p,b) appearing in $\mathsf{BB}_{\mathsf{init}}$, where (id, v_0, v_1, p, b) is the corresponding element in V (in that case, $\pi(i)$ is the index of this element in V).
- $\overline{\mathsf{BB}}[i] = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}[i])$, which $\mathcal{B}$ gets from its call to $\mathcal{O}_{\mathsf{d}}$, if $\mathsf{BB}[i]$ does not appear in $\mathsf{BB}_{\mathsf{init}}$.

At this point, we have $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))$, except with negligible probability. Indeed, for any i:

- Either $\mathsf{BB}[i]$ does not appear in $\mathsf{BB}_{\mathsf{init}}$, and, by definition, $\pi(i) = \mathsf{BB}[i]$. Hence $\overline{\pi}(\mathsf{BB}_0)[i] = \mathsf{BB}[i]$. Then $\overline{\mathsf{BB}}[i] = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}[i])$ holds by construction of $\overline{\mathsf{BB}}$.
- Or $\mathsf{BB}_{\mathsf{init}}[j] = (id, \mathsf{BB}[i])$ for some j, id, and, by definition, $\pi(i) = j$. Hence $\overline{\pi}(\mathsf{BB}_0)[i] = \mathsf{BB}[i]$. Let $(id, v_0, v_1, \mathsf{BB}[i])$ denote $V[j]$. By construction by the $\mathcal{O}\mathsf{voteLR}$ oracle, $\mathsf{BB}_0[j] = (id, (p,b))$, where (p,b) is a ballot created by calling $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$. By construction of $\overline{\mathsf{BB}}$, $\overline{\mathsf{BB}}[i] = (id, v_0)$. Hence,

$$\begin{array}{rl} & \mathbb{P}(\overline{\mathsf{BB}}[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}[i]))) \\ \leq & \mathbb{P}((\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda);\ \mathsf{U} \leftarrow \mathsf{Register}(1^\lambda, \mathcal{I}); \\ & \quad (s, p, b) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0);\ \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) \neq (id, v_0)) \end{array}$$

  which is negligible by strong consistency.

Since $\overline{\mathsf{BB}}$ has as many elements as BB, it is of polynomial size (bounded by the running time of $\mathcal{A}$). Hence, $\mathbb{P}(\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))) = \mathbb{P}(\exists i.\ \overline{\mathsf{BB}}[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0)[i]))$ is also negligible. We write

$$p^\beta_3(\lambda) = \mathbb{P}(\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0)) \text{ in } \mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{B},\mathcal{V}}).$$

$\mathcal{B}$ then computes $\rho(\overline{\mathsf{BB}})$, and shows this result to $\mathcal{A}$. $\mathcal{A}$ finally outputs a bit β', that $\mathcal{B}$ returns.

We now argue that $\rho(\overline{\mathsf{BB}})$ is indeed the result $\mathcal{A}$ would have been shown in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$.

- If $\beta = 1$, $\mathsf{BB}_{\mathsf{init}} = \mathsf{BB}_1$, and thus $\pi = \textsc{Recover}^{\mathsf{del},\mathsf{reorder}}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$. As previously explained, we then have

$$\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\textsc{Recover}^{\mathsf{del},\mathsf{reorder}}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})}(\mathsf{BB}_0)),$$

  that is, $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$, where BB' is the board computed using $\textsc{Recover}^{\mathsf{del},\mathsf{reorder}}$ in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}$. By definition of $\mathcal{O}\mathsf{tally}'$, $\rho(\overline{\mathsf{BB}})$ is thus the election result $\mathcal{A}$ sees in this game.

- If $\beta = 0$, $\mathsf{BB}_{\mathsf{init}} = \mathsf{BB}_0$, and thus $\pi = \textsc{Recover}_{\mathsf{U}}^{\mathsf{del,reorder}}(\mathsf{BB}_0, \mathsf{BB})$. Recall that, since all the verifications have succeeded, all the ballots in BB_0 produced by honest voters who check correctly occur in BB. Therefore, it is clear from the definition of $\textsc{Recover}^{\mathsf{del,reorder}}$ that $\overline{\pi}(\mathsf{BB}_0) = \mathsf{BB}$. As previously explained, we then have $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})$, and, by definition of $\mathcal{O}\mathsf{tally}'$, $\rho(\overline{\mathsf{BB}})$ is thus the election result $\mathcal{A}$ sees in this $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},0}$.

As we argued, except if $\mathsf{BB}_{\mathsf{init}}$ contains duplicate ballots, or π does not have the expected form, or $\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}\mathsf{BB}_0)$, $\mathcal{A}$ is run until the end of its execution by $\mathcal{B}$ if and only if it would also reach the end of the game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{mb\text{-}BPRIV1},\beta}$; and $\mathcal{A}$ run by $\mathcal{B}$ has the same view it would have in the corresponding execution of $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},\beta}$. Hence, except in those three cases, $\mathcal{B}$ returns 1 in $\mathsf{Exp}_{\mathcal{B}}^{\mathsf{NM},\beta}$ if and only if $\mathcal{A}$ returns 1 in the corresponding execution of $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{mb\text{-}BPRIV1},\beta}$. Thus, for any β, if $p^\beta(\lambda)$ denote $p_1^\beta(\lambda) + p_2^\beta(\lambda) + p_3^\beta(\lambda)$,

$$|\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},\beta}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},\beta}(\lambda) = 1)| \leq p^\beta(\lambda).$$

Therefore

$$\begin{array}{rl} & |\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},1}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},0}(\lambda) = 1)| \\ & + |\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},1}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},1}(\lambda) = 1)| \\ & + |\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},1}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},1}(\lambda) = 1)| + p^0(\lambda) + p^1(\lambda), \end{array}$$

which implies that, assuming strong consistency holds, and that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{NM}}$, no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$.

$\underline{\mathsf{Exp}^{\mathbf{mb\text{-}BPRIV1}} \wedge \mathsf{Exp}^{\mathsf{SC}} \Rightarrow \mathsf{Exp}^{\mathbf{mb\text{-}BPRIV2}}.}$

We now show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. Consider the same $\mathcal{A}$ playing $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$ instead. $\mathcal{A}$ has the same view in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},\beta}$ as in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV2},\beta}$, except if the result r returned by the Tally algorithm differs from ρ applied to the extractions of the board, which happens only with negligible probability by the second point of the strong consistency assumption.

More precisely, consider an adversary $\mathcal{B}$ playing $\mathsf{Exp}^{\mathsf{SC}}$, that runs $\mathcal{A}$ internally, simulating its execution in game $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV2},0}$. $\mathcal{B}$ answers any call to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ made by $\mathcal{A}$ by running $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$, and returning the generated ballot to $\mathcal{A}$. When $\mathcal{A}$ produces a board BB, $\mathcal{B}$ returns this board.

When $\beta = 0$, $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},0}$ and $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV2},0}$ give $\mathcal{A}$ the same view, and hence return the same result, except if $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})) \neq r$, where $(r, \Pi) = \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$. That is, except if $\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{SC}}$ returns 1. Therefore

$$|\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV2},0}(\lambda) = 1)| \leq \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{SC}}(\lambda) = 1).$$

Similarly, consider an adversary $\mathcal{C}$ playing $\mathsf{Exp}^{\mathsf{SC}}$, that runs $\mathcal{A}$ internally, simulating its execution in game $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV2},1}$. $\mathcal{C}$ answers any call to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ made by $\mathcal{A}$ by running $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$ and $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_1)$, storing the generated ballots in boards BB_0 and BB_1, and returning the ballot for v_1 to $\mathcal{A}$. When $\mathcal{A}$ produces a board BB, $\mathcal{C}$ computes $\mathsf{BB}' = \overline{\textsc{Recover}_{\mathsf{U}}^{\mathsf{del,reorder}}(\mathsf{BB}_1, \mathsf{BB})}(\mathsf{BB}_0)$, and returns this board.

When $\beta = 1$, $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}$ give $\mathcal{A}$ the same view, and hence return the same result, except if $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')) \neq r$, where $(r, \Pi) = \mathsf{Tally}(\mathsf{BB}', \mathsf{sk})$. That is, except if $\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}$ returns 1. Therefore

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \leq \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}(\lambda) = 1).$$

Therefore:

$$\begin{array}{rl} & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & +|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & +|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & +\mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) + \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}(\lambda) = 1), \end{array}$$

which implies, by the strong consistency assumption, that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$.

$\underline{\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}} \wedge \textbf{ different random oracles } \Rightarrow \mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}.}$

We now show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. Consider an adversary $\mathcal{B}$ against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$, that runs $\mathcal{A}$ internally. During the voting and verification phases, $\mathcal{B}$ answers $\mathcal{A}$'s oracle queries by making the same calls to its oracles, once $\mathcal{A}$ returns a board BB, $\mathcal{B}$ returns this same board. $\mathcal{B}$ also runs its own random oracle, that is made available to $\mathcal{A}$, to simulate the random oracle used for the proof of correct tallying that $\mathcal{A}$ expects to receive. This is made possible by the assumption that this random oracle is not used in any other part of the protocol. Once $\mathcal{A}$ calls $\mathcal{O}\mathsf{tally}'''$, $\mathcal{B}$ calls $\mathcal{O}\mathsf{tally}''$ and receives a result r. Manipulating the random oracle it runs, $\mathcal{B}$ then produces a proof $\Pi = \mathsf{SimProof}(r, \mathsf{BB})$, and returns (r, Π) to $\mathcal{A}$. When $\mathcal{A}$ makes its guess regarding β, $\mathcal{B}$ returns the same guess.

It is clear that $\mathcal{A}$ as run by $\mathcal{B}$ has the same view it would have in game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},\beta}_{\mathcal{A},\mathcal{V}}$. Hence

$$\begin{array}{l} |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| = \\ |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \end{array}$$

is negligible, which proves no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$.

$\underline{\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}} \wedge \mathsf{Exp}^{\mathsf{ZK}} \Rightarrow \mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}.}$

Finally we show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$. Consider $\mathcal{A}$ playing $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$ instead.

It is clear that, when $\beta = 1$, $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}}$ are identical. Hence

$$\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}} = 1).$$

When $\beta = 0$, the outputs of $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}$ can only differ if $\mathcal{A}$ is able to distinguish the real proof of correct tallying Π returned by $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$ from the simulated proof $\mathsf{SimProof}(r, \mathsf{BB})$.

More precisely, let $\mathcal{B}$ be an adversary against $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{ZK},\beta'}$. $\mathcal{B}$ runs $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV},0}$ internally. That is, $\mathcal{B}$ generates its own sets of credentials U, using the $\mathsf{Register}$ algorithm, and runs $\mathcal{A}$, answering calls to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ by computing $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$. $\mathcal{B}$ obtains a board BB from $\mathcal{A}$, on which it performs the validity check using $\mathsf{ValidBoard}$. If this check fails, $\mathcal{B}$ answers 0. Otherwise, it lets $\mathcal{A}$ call its verification oracle, using Verify to answer $\mathcal{A}$'s queries. Again, if $\mathcal{A}$ does not make all voters in $\mathsf{H}_{\mathsf{check}}$ verify, $\mathcal{B}$ answers 0. If some verifications fail, $\mathcal{B}$ asks $\mathcal{A}$ for its guess regarding the bit β', and returns it. Otherwise, $\mathcal{B}$ returns the board BB, and obtains (r, Π_β), where r is the result of the tally, and $\Pi_{\beta'}$, depending on β', is either the real proof Π_0 computed by $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$, or the simulated proof $\Pi_1 = \mathsf{SimProof}(r, \mathsf{BB})$. $\mathcal{B}$ continues to run $\mathcal{A}$, answering (r, Π_β) when $\mathcal{A}$ calls $\mathcal{O}\mathsf{tally}$. Finally $\mathcal{A}$ makes a guess regarding β', that $\mathcal{B}$ returns as its output.

It is clear that, if $\beta' = 0$, $\mathcal{A}$ as simulated by $\mathcal{B}$ in $\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},0}$ has the same view it would have in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV},0}$. Hence, $\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},0} = 1) = \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV},0} = 1)$.

Similarly if $\beta' = 1$, $\mathcal{A}$ simulated by $\mathcal{B}$ in $\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},1}$ has the same view it would have in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV3},0}$. Hence, $\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},1} = 1) = \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV3},0} = 1)$.

Thus,

$$\begin{aligned}
& |\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV},1}(\lambda) = 1)| \\
= \; & |\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV3},1}(\lambda) = 1)| \\
\leq \; & |\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},1}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},1}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV3},1}(\lambda) = 1)| \\
= \; & |\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{ZK},1}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV3},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV3},1}(\lambda) = 1)|,
\end{aligned}$$

which implies that, assuming that the zero-knowledge property holds, no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$. □

B.2.6 Ideal functionality corresponding to Recover$^{\mathsf{del,reorder}}$

Recall the predicate $P^{\mathsf{del,reorder}}$ that is checked by the functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$:

$P^{\mathsf{del,reorder}}(L, f) = \top$ iff:

- f keeps the votes of all voters who check:
  $\forall i.\, \forall id \in \mathsf{H}_{\mathsf{check}}.\, \forall v.\, L[i] = (id, v) \implies \exists j.\, f(j) = i.$
- and no honest votes are modified by f:
  $\forall i, id, v.\, f(i) = (id, v) \implies id \in \mathsf{D}.$

Theorem B.14. *The* RECOVER$^{\mathsf{del,reorder}}$ *algorithm defined above is compatible with the predicate* $P^{\mathsf{del,reorder}}$.

Proof. Indeed, consider an adversary $\mathcal{A}$, that plays the game $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{comp},P^{\mathsf{del,reorder}},\text{RECOVER}^{\mathsf{del,reorder}}}$. Following this game, let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, $\mathsf{U} \leftarrow \mathsf{Register}(1^\lambda, \mathcal{I})$, $\mathsf{CU} \leftarrow [\mathsf{U}[id] | id \in \mathsf{D}]$, and $\mathsf{PU} \leftarrow [\mathsf{Pub}(\mathsf{U}[id]) | id \in \mathcal{I}]$. $\mathcal{A}$ has access to the $\mathcal{O}\mathsf{vote}$ oracle, to generate honest ballots, that get stored in a board BB_1. For each $(id, (p, b)) \in \mathsf{BB}_1$, by construction, (p, b) was constructed by calling $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$ for some v. Let BB be the board returned by $\mathcal{A}$ in the game. Note that, by the non-malleability assumption, following the same reasoning as in the previous proof, BB_1 contains no duplicate ballots, except with negligible probability. If BB is not valid, or BB_1 does not contain at least one entry for each $id \in \mathsf{H}_{\mathsf{check}}$, $\mathcal{A}$ loses the game.

Otherwise, let $\pi = \textsc{Recover}_{\mathsf{U}}^{\mathsf{del,reorder}}(\mathsf{BB}_1, \mathsf{BB})$. Let us show that π is compatible $P^{\mathsf{del,reorder}}$ *w.r.t.* $\mathsf{sk}, \mathsf{U}, L_{id}$, except with negligible probability. Assume BB_1 contains no duplicate ballots, which, as explained, holds with overwhelming probability.

Let L be a list of elements (id, v), such that $[id|(id, v) \in L] = \mathsf{L}_{id}$. We show that $P^{\mathsf{del,reorder}}(L, \mathsf{mod}_{\mathsf{sk,U}}(\pi)) = \top$.

Let $L' = [i|\exists p, b.\ \mathsf{BB}_1[i] = (id, (p, b)) \ \wedge\ id \in \mathsf{H}_{\mathsf{check}} \ \wedge\ (p, b) \notin \mathsf{BB}]$, the list of the positions in BB_1 of ballots belonging to voters in $\mathsf{H}_{\mathsf{check}}$ that do not occur in BB. Let also LL be the list constructed by the following process:

$LL \leftarrow []$;
for $(p, b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $LL \leftarrow LL \parallel j$
    (by assumption on BB_1, this j is unique)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ **then**
    $LL \leftarrow LL \parallel \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$.

By definition, $\mathsf{mod}_{\mathsf{sk,U}}(\pi)$ is $\lambda i.LL'''[i]$, where LL''' is the list obtained obtained by removing all $\perp$ elements from $LL'' = LL \parallel L'$.

We have to show that $P^{\mathsf{del,reorder}}(L, \mathsf{mod}_{\mathsf{sk,U}}(\pi)) = \top$. That is, that

1. No honest votes are modified by $\mathsf{mod}_{\mathsf{sk,U}}(\pi)$:

$$\forall i.\ \forall (id, v).\ \mathsf{mod}_{\mathsf{sk,U}}(\pi)[i] = (id, v) \implies id \in \mathsf{D}.$$

   Let i, id, v be such that $\mathsf{mod}_{\mathsf{sk,U}}(\pi)[i] = (id, v)$. Hence $LL[j] = (id, v)$ for some j. Thus, by construction of LL, $\mathsf{BB}[k] = (p, b)$ for some k, p, b such that (p, b) does not occur in BB_1, $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$, and $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}[j]) = (id, v)$. Therefore $id \in \mathsf{D}$.

2. $\mathsf{mod}_{\mathsf{sk,U}}(\pi)$ keeps the votes of all voters who check:

$$\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall v.\ L[i] = (id, v) \implies \exists j.\ \mathsf{mod}_{\mathsf{sk,U}}(\pi)[j] = i.$$

   Let i, $id \in \mathsf{H}_{\mathsf{check}}$, v such that $L[i] = (id, v)$. By definition of L, $L_{id}[i] = id$. Then by definition of L_{id}, there exist p, b such that $\mathsf{BB}_1[i] = (id, (p, b))$. Hence, by definition of L', either $(p, b) \in \mathsf{BB}$, or $i \in L'$. In the first case, by construction of LL, $i \in LL$, and thus $\exists j.\ \mathsf{mod}_{\mathsf{sk,U}}(\pi)(j) = i$. In the second case, since, by construction of LL'', L' is a sublist of LL'', we have $i \in LL''$ and hence $\exists j.\ \mathsf{mod}_{\mathsf{sk,U}}(\pi)(j) = i$. In any case, the claim holds, which concludes the proof.

□

B.2.7 Conclusion on Belenios and Civitas without revote

Theorem B.15. *Assume Belenios, with no revote allowed, is strongly consistent, has non-malleable ballots and credentials, that the proofs of correct tallying are zero-knowledge, and that different random oracles are used for these proofs and the rest of the protocol. Then Belenios, securely implements ideal functionalities* $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ *and* $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$ *against environments that make each voter vote at most once.*

Under the same assumptions, Civitas also implements these three functionalities.

Proof.

- It directly follows from Theorems 12, B.13, and B.14 that, under these assumptions, Belenios and Civitas implement the functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$.

- It is also clear that $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ is stronger than $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$, in that it gives less power to the simulator. Indeed these two functionalities are the same, except that $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$ allows more modification functions from $\mathcal{S}$ than $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$. Any simulator $\mathcal{S}$ running with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ can thus be accurately simulated by a simulator $\mathcal{S}'$ running with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$, that runs $\mathcal{S}$ internally, checks whether its proposed modification function f is accepted by $P^{\mathsf{del,reorder}}$, then submits f if so and blocks the result otherwise. (Note $\mathcal{S}'$ can check this condition using only f and the sequence of the *id*s of voters who submitted votes, but does not need to know the votes themselves, by definition of $P^{\mathsf{del,reorder}}$.) Therefore, any voting scheme that securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ also securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$, which proves the claim that Belenios and Civitas implement this functionality.

- In addition, it is clear by carefully examining the ideal execution that when only considering environment $\mathcal{E}$ that do not make voters revote, functionalities $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ are identical. Indeed, in that case, the list L that $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ keep respectively in $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho))$ and $\mathsf{idealexec}(\mathcal{E}||\mathcal{S}||\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho))$ only contains at most one entry for each honest *id*. For such a list L, we have for any f that $P^{\mathsf{del}}(L,f) = P^{\mathsf{del,reorder}}(L,f)$. Since the predicates P^{del} and $P^{\mathsf{del,reorder}}$ are the only difference between the two functionalities, they are indeed equivalent in the case of environments that do not make voters revote, which proves the claim that Civitas and Belenios securely implement $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$.

□

B.3 Case study: Belenios with revote

We show here that, under reasonable assumptions, Belenios with revote is mb-BPRIV *w.r.t.* $\textsc{Recover}^{\mathsf{del}'}$ (defined below), and that this implies it realises $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$. Consequently it also realises the weaker $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$.

B.3.1 Notations

Compared to the "no revote" case, the verification algorithm differs. $\mathsf{Verify}(id, s, \mathsf{BB})$ now checks whether the last ballot recorded in s, *i.e.* the last ballot generated by id, is the last ballot that appears next to the public key of id in BB, or, equivalently when the board is valid, the last ballot signed with id's key in BB.

Since revote is now allowed, the validity algorithm ValidBoard also changes: it no longer rejects a board that contains several ballots associated with public credentials encrypting the same private credential. However it still checks all the proofs and signatures on all ballots.

B.3.2 Recovery

We consider the following recovery algorithm:

$\textsc{Recover}^{\mathsf{del}'}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$

$L \leftarrow []; L_{\mathsf{cast}} \leftarrow []; I \leftarrow [];$
$\mathsf{BB}_{\mathsf{revote}} \leftarrow \mathsf{BB}$ where only the last (p, b) is kept for each public key p
for $(p, b) \in \mathsf{BB}_{\mathsf{revote}}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $L \leftarrow L \parallel j; I[id] \leftarrow j;$
    (in case several such j exist, pick the first one)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ **then**
    $L \leftarrow L \parallel (p, b); L_{\mathsf{cast}} \leftarrow L_{\mathsf{cast}} \parallel (p, b);$
$\forall id \in \mathsf{H}_{\mathsf{check}}.\ LL_{id} \leftarrow [i | \exists p, b.\ \mathsf{BB}_1[i] = (id, (p, b))]$
if $\forall id \in \mathsf{H}_{\mathsf{check}}.\ I[id] =$ the last element of LL_{id} **then**
  $L' \leftarrow L$
  for $id \in \mathsf{H}_{\mathsf{check}}$ **do**
    insert LL_{id} without its last element right before $I[id]$ in L'
  return $(\lambda i.\ L'[i])$
else
  $L' \leftarrow [1 \ldots |\mathsf{BB}_1|] \parallel L_{\mathsf{cast}};$
  return $(\lambda i.\ L'[i])$

B.3.3 Assumptions

We assume, as before, that the ballots as well as the credentials are non malleable, *i.e.* that for all adversary $\mathcal{A}$,

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|$$

and $|\mathbb{P}(\mathsf{Exp}^{\mathsf{CNM}}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|$ are negligible in λ.

As before, we also assume that the voting scheme is strongly consistent, that the proof of correct tallying has the zero-knowledge property, and that the proof of correct tallying and the rest of the protocol use different random oracles.

In addition, we will assume that the revote policy encoded by the counting function ρ is to keep each voter's last vote. That is, for any list L of pairs (id, v), if L' is the list obtained from L by removing each element except the last pair for each distinct id, then $\rho(L) = \rho(L')$.

B.3.4 Belenios with revote is mb-BPRIV

Theorem B.16. *Assume Belenios, with revote allowed, is strongly consistent, has non-malleable ballots and credentials, that the proofs of correct tallying are zero-knowledge, that different random oracles are used for these proofs and the rest of the protocol, and that ρ encodes the "last vote counts" revote policy.*

Then Belenios is mb-BPRIV w.r.t. $\textsc{Recover}^{\mathsf{del}'}$.

Proof. As for the "no revote" case, we will consider a sequence of games that differ from $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover}^{\mathsf{del}'}}$ by the way the result the adversary gets to see is computed. These games are the same as before, except they use $\textsc{Recover}^{\mathsf{del}'}$ instead of $\textsc{Recover}^{\mathsf{del},\mathsf{reorder}}$:

- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ is not shown the tally of the board/the recovered board by $\mathcal{O}\mathsf{tally}$, $\mathcal{O}\mathsf{tally}'$ instead computes the result using ρ and the $\mathsf{extract}$ function, without any proof of correct tallying.
- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ has access to $\mathcal{O}\mathsf{tally}''$, who computes the result of the election using the Tally algorithm, but still no proof of correct tallying.
- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ is shown by $\mathcal{O}\mathsf{tally}'''$ the result of the election computed by the Tally algorithm, but the proof of correct tallying for the board provided by $\mathcal{A}$ is simulated on both sides.

The structure of the proof is similar to the "no revote" case. We first show that if no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{NM}}$ or $\mathsf{Exp}^{\mathsf{CNM}}$, then no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$. Using the strong consistency assumption, we then show that this implies no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. We then show, using the assumption that the random oracles used for the proof of correct tallying and the remainder of the protocol are different, that this implies no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. From there, using the zero-knowledge assumption on the proof of correct tallying, we show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$.

$\underline{\mathsf{Exp}^{\mathsf{NM}} \wedge \mathsf{Exp}^{\mathsf{CNM}} \Rightarrow \mathsf{Exp}^{\mathbf{mb\text{-}BPRIV1}}.}$

Let $\mathcal{A}$ be an attacker that wins the $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},}_{\mathcal{V}}$ game.

We construct an attacker $\mathcal{B}$ against $\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{V}}$. $\mathcal{B}$ is given access to pk and U. It runs $\mathcal{A}$ internally, simulating the oracle calls as follows. When $\mathcal{A}$ calls $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$, for some $id \in \mathsf{H}$, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{c}}(id, \mathsf{U}[id], v_0, v_1)$ and obtains some (p, b). $\mathcal{B}$ records $(id, (p, b))$ in a list $\mathsf{BB}_{\mathsf{init}}$, and stores (id, v_0, v_1, p, b) in a list V. $\mathcal{B}$ then returns (p, b) to $\mathcal{A}$. At some point, $\mathcal{A}$ returns to $\mathcal{B}$ some board BB.

Note that

- By construction, during the execution, the list $[(p, b)|(id, (p, b)) \in \mathsf{BB}_{\mathsf{init}}]$ of the ballots in $\mathsf{BB}_{\mathsf{init}}$ (kept by $\mathcal{B}$) is always equal to the list L in the game $\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{V}}$ played by $\mathcal{B}$.
- It is also clear by examining the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A}}$ that, up to this point, $\mathcal{A}$ has been accurately simulated, and has the same view it would have in game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A}}$.
- By construction of the $\mathcal{O}\mathsf{voteLR}$ oracle, the list $\mathsf{BB}_{\mathsf{init}}$ is also equal to the list BB_β in (the corresponding execution of) the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A}}$ for $\mathcal{A}$.
- We can also note that $\mathsf{BB}_{\mathsf{init}}$ contains no duplicate ballots (except with a negligible probability $p_1^\beta(\lambda)$). Indeed, $\mathsf{BB}_{\mathsf{init}}$ containing duplicate ballots would imply that two oracle calls $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$ and $\mathsf{Vote}(\mathsf{pk}, id', \mathsf{U}[id'], v')$ produced the same ballot. If this happened with non-negligible probability, by submitting $\mathcal{O}_{\mathsf{c}}(id, \mathsf{U}[id], v, v'')$ followed by $\mathcal{O}_{\mathsf{c}}(id', \mathsf{U}[id'], v', v''')$ (for some $v'' \neq v'''$) to the encryption oracle, and checking whether the two resulting ballots are equal, an adversary would win the non-malleability game. The two ballots obtained on the right would indeed be different, except with negligible probability, by strong consistency (as they contain different votes).

Once $\mathcal{A}$ has returned BB, $\mathcal{B}$ checks (in V) that all voters in $\mathsf{H}_{\mathsf{check}}$ have voted, and halts if not. $\mathcal{B}$ also checks whether $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$. If not, $\mathcal{B}$ directly asks $\mathcal{A}$ to guess the bit β', and returns $\mathcal{A}$'s guess.

Following the structure of game $\mathsf{Exp}^{\text{mb-BPRIV1}}$, $\mathcal{B}$ then lets $\mathcal{A}$ perform the verifications for the honest voters. That is, when $\mathcal{A}$ calls $\mathcal{O}\mathsf{verify}_{\mathsf{BB}}(id)$, $\mathcal{B}$ retrieves the last entry (id, v_0, v_1, p, b) for id in V, and checks that the last $(p', b') \in \mathsf{BB}$ such that $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p') = id$ is indeed (p, b). Once $\mathcal{A}$ is done with the verification phase, $\mathcal{B}$ checks that each $id \in \mathsf{H}_{\mathsf{check}}$ has verified (and halts otherwise). If any of the verifications have failed, $\mathcal{B}$ directly asks $\mathcal{A}$ to guess the bit β', and returns $\mathcal{A}$'s guess.

If all verifications are successful, $\mathcal{B}$ will simulate the tallying phase to $\mathcal{A}$. To do this, $\mathcal{B}$ first computes $\pi = \textsc{Recover}^{\mathsf{del}'}{}_{\mathsf{U}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB})$.

Note, at this point, that since all checks succeeded, π is simpler than in the general case. Indeed, the verifications guarantee that the last ballot cast by each $id \in \mathsf{H}_{\mathsf{check}}$ is the last ballot to occur for id's key in BB. Therefore, the test "**if** $\forall id \in \mathsf{H}_{\mathsf{check}}.\ I[id] =$ the last element of LL_{id}" performed by $\textsc{Recover}^{\mathsf{del}'}{}_{\mathsf{U}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB})$ succeeds.

Let $\mathsf{BB}_{\mathsf{revote}}$ be the board obtained by applying the revote policy to BB, *i.e.* keeping only the last ballot for each public key. From the definition of $\textsc{Recover}^{\mathsf{del}'}$, it appears clearly that π is in fact equal to $\textsc{Recover}^{\mathsf{del}'}{}_{\mathsf{U}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB}_{\mathsf{revote}})$. Then, consider L the list of length $|\mathsf{BB}_{\mathsf{revote}}|$ such that for all i:

- $L[i] = j$ if j is the index of the first (and only) occurrence of $\mathsf{BB}'[i]$ in $\mathsf{BB}_{\mathsf{init}}$,
- or $L[i] = \mathsf{BB}_{\mathsf{revote}}[i]$ if no such index exists. Note that, in that case, we have $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ (where p is the public key in $\mathsf{BB}_{\mathsf{revote}}[i]$), except with negligible probability $p_2^{\beta}(\lambda)$: otherwise an adversary could win game $\mathsf{Exp}^{\mathsf{CNM}}$ by simulating $\mathcal{A}$ and returning $\mathsf{BB}_{\mathsf{revote}}$.

By definition of $\textsc{Recover}^{\mathsf{del}'}$, π is the function $\lambda i.\ L'[i]$, where L' is obtained from L by inserting, for each $id \in \mathsf{H}_{\mathsf{check}}$, the list LL_{id} of all indices of id's ballots in $\mathsf{BB}_{\mathsf{init}}$ before the index of id's last ballot. We will also denote π_{revote} the function $\lambda i.\ L[i]$.

To continue the simulation of $\mathcal{A}$, $\mathcal{B}$ must then provide $\mathcal{A}$ with the result of the election, to answer $\mathcal{A}$'s calls to $\mathcal{O}\mathsf{tally}'$. $\mathcal{B}$ asks for the decryption of the list of all (p, b) such that $\exists j.\ \pi(j) = (p, b)$. That is, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{d}}([(p, b) | \exists j.\ \pi(j) = (p, b)])$. This call to the decryption oracle is allowed: indeed, by the description of π above, a ballot such that $\pi(j) = (p, b)$ cannot occur in an element of $\mathsf{BB}_{\mathsf{init}}$, and hence is not in L.

$\mathcal{B}$ then constructs a list $\overline{\mathsf{BB}}$ containing the votes in $\overline{\pi}\mathsf{BB}_0$ in clear, BB_0 being the board of the left ballots, maintained by $\mathcal{O}\mathsf{voteLR}$ in $\mathsf{Exp}^{\text{mb-BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$. This is done by retrieving the vote from the list V for ballots coming from the encryption oracle, and using the decryption oracle for the others. Formally this list is obtained by, for all i:

- $\overline{\mathsf{BB}}[i] = (id, v_0)$ if $\pi(i) = j$, where $\mathsf{V}[j] = (id, v_0, v_1, p, b)$,
- $\overline{\mathsf{BB}}[i] = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$, which $\mathcal{B}$ gets from its call to $\mathcal{O}_{\mathsf{d}}$, if $\pi(i) = (p, b)$.

At this point, we have $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))$, except with negligible probability. Indeed, for any i:

- Either $\pi(i) = (p, b)$ for some p, b. Hence $\overline{\pi}(\mathsf{BB}_0)[i] = (p, b)$, and $\overline{\mathsf{BB}}[i] = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$ holds by construction of $\overline{\mathsf{BB}}$.
- Or $\pi(i) = j$ for some j. Then let (id, v_0, v_1, p, b) denote $\mathsf{V}[j]$. By construction of $\overline{\mathsf{BB}}$, $\overline{\mathsf{BB}}[i] = (id, v_0)$. In addition, bu definition of $\overline{\pi}$, $\overline{\pi}(\mathsf{BB}_0)[i] = (p, b)$. By construction

by the $\mathcal{O}$voteLR oracle, $\mathsf{BB}_0[j] = (id, (p, b))$, and (p, b) is a ballot created by calling $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$. Hence,

$$\begin{array}{rl} & \mathbb{P}(\overline{\mathsf{BB}}[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}[i]))) \\ \leq & \mathbb{P}((\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda);\ \mathsf{U} \leftarrow \mathsf{Register}(1^\lambda, \mathcal{I}); \\ & \quad (s, p, b) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0);\ \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) \neq (id, v_0)) \end{array}$$

which is negligible by strong consistency.

Since $\overline{\mathsf{BB}}$ has as many elements as BB, it is of polynomial size (bounded by the running time of $\mathcal{A}$). Hence, $\mathbb{P}(\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))) = \mathbb{P}(\exists i.\ \overline{\mathsf{BB}}[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0)[i]))$ is also negligible. We write

$$p_3^\beta(\lambda) = \mathbb{P}(\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0)) \text{ in } \mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{B},\mathcal{V}}).$$

$\mathcal{B}$ then computes $\rho(\overline{\mathsf{BB}})$, and shows this result to $\mathcal{A}$. $\mathcal{A}$ finally outputs a bit β', that $\mathcal{B}$ returns.

We now argue that $\rho(\overline{\mathsf{BB}})$ is indeed the result $\mathcal{A}$ would have been shown in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$.

- If $\beta = 1$, $\mathsf{BB}_{\mathsf{init}} = \mathsf{BB}_1$, and thus $\pi = \textsc{Recover}^{\mathsf{del}'_{\mathsf{U}}}(\mathsf{BB}_1, \mathsf{BB})$. As previously explained, we then have
$$\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\textsc{Recover}^{\mathsf{del}'_{\mathsf{U}}}(\mathsf{BB}_1, \mathsf{BB})(\mathsf{BB}_0)}),$$
that is, $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$, where BB' is the board computed using $\textsc{Recover}^{\mathsf{del}'}$ in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}$. By definition of $\mathcal{O}\mathsf{tally}'$, $\rho(\overline{\mathsf{BB}})$ is thus the election result $\mathcal{A}$ sees in this game.

- If $\beta = 0$, $\mathsf{BB}_{\mathsf{init}} = \mathsf{BB}_0$, and thus $\pi = \textsc{Recover}^{\mathsf{del}'_{\mathsf{U}}}(\mathsf{BB}_0, \mathsf{BB})$. By definition of $\mathcal{O}\mathsf{tally}'$, $\mathcal{A}$ receives the result $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}))$ in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}$. By assumption, ρ follows the "last vote counts" revote policy. Thus $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})) = \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}_{\mathsf{revote}}))$. In addition, for the same reason, it is clear that
$$\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))) = \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi_{\mathsf{revote}}}(\mathsf{BB}_0))).$$
That is, $\rho(\overline{\mathsf{BB}}) = \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi_{\mathsf{revote}}}(\mathsf{BB}_0)))$.

  Recall that, since all the verifications have succeeded, for each honest voter who checks $id \in \mathsf{H}_{\mathsf{check}}$, all the last ballot in BB_0 produced by id occurs in $\mathsf{BB}_{\mathsf{revote}}$. Therefore, it is clear from the definition of π_{revote} that $\overline{\pi}(\mathsf{BB}_0) = \mathsf{BB}_{\mathsf{revote}}$. Hence, $\rho(\overline{\mathsf{BB}}) = \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}_{\mathsf{revote}})) = \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}))$ is indeed the election result $\mathcal{A}$ sees in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}$.

As we argued, except if

- $\mathsf{BB}_{\mathsf{init}}$ contains duplicate ballots ($p_1^\beta(\lambda)$),
- or π does not have the expected form ($p_2^\beta(\lambda)$),
- or $\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}\mathsf{BB}_0)$ ($p_3^\beta(\lambda)$),

$\mathcal{A}$ is run until the end of its execution by $\mathcal{B}$ if and only if it would also reach the end of the game $\mathsf{Exp}_{\mathcal{A}}^{\text{mb-BPRIV1},\beta}$; and $\mathcal{A}$ run by $\mathcal{B}$ has the same view it would have in the corresponding execution of $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},\beta}$. Hence, except in those three cases, $\mathcal{B}$ returns 1 in $\mathsf{Exp}_{\mathcal{B}}^{\mathsf{NM},\beta}$ if and only if $\mathcal{A}$ returns 1 in the corresponding execution of $\mathsf{Exp}_{\mathcal{A}}^{\text{mb-BPRIV1},\beta}$. Thus, for any β, if $p^\beta(\lambda)$ denotes $p_1^\beta(\lambda) + p_2^\beta(\lambda) + p_3^\beta(\lambda)$,

$$|\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},\beta}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},\beta}(\lambda) = 1)| \leq p^\beta(\lambda).$$

Therefore

$$\begin{aligned}
& |\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},1}(\lambda) = 1)| \\
\leq\ & |\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},0}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},1}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},1}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},1}(\lambda) = 1)| \\
\leq\ & |\mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},1}(\lambda) = 1)| + p^0(\lambda) + p^1(\lambda),
\end{aligned}$$

which implies that, assuming strong consistency holds, and that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{NM}}$, no adversary has a non-negligible advantage in $\mathsf{Exp}^{\text{mb-BPRIV1}}$.

The remaining steps of the proof are identical to the "no revote" case.

$\underline{\mathsf{Exp}^{\textbf{mb-BPRIV}1} \wedge \mathsf{Exp}^{\mathsf{SC}} \Rightarrow \mathsf{Exp}^{\textbf{mb-BPRIV}2}.}$

We now show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\text{mb-BPRIV2}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\text{mb-BPRIV2}}$. Consider the same $\mathcal{A}$ playing $\mathsf{Exp}^{\text{mb-BPRIV1}}$ instead. $\mathcal{A}$ has the same view in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},\beta}$ as in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV2},\beta}$, except if the result r returned by the Tally algorithm differs from ρ applied to the extractions of the board, which happens only with negligible probability by the second point of the strong consistency assumption.

More precisely, consider an adversary $\mathcal{B}$ playing $\mathsf{Exp}^{\mathsf{SC}}$, that runs $\mathcal{A}$ internally, simulating its execution in game $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV2},0}$. $\mathcal{B}$ answers any call to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ made by $\mathcal{A}$ by running $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$, and returning the generated ballot to $\mathcal{A}$. When $\mathcal{A}$ produces a board BB, $\mathcal{B}$ returns this board.

When $\beta = 0$, $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},0}$ and $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV2},0}$ give $\mathcal{A}$ the same view, and hence return the same result, except if $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})) \neq r$, where $(r, \Pi) = \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$. That is, except if $\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{SC}}$ returns 1. Therefore

$$|\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV2},0}(\lambda) = 1)| \leq \mathbb{P}(\mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{SC}}(\lambda) = 1).$$

Similarly, consider an adversary $\mathcal{C}$ playing $\mathsf{Exp}^{\mathsf{SC}}$, that runs $\mathcal{A}$ internally, simulating its execution in game $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV2},1}$. $\mathcal{C}$ answers any call to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ made by $\mathcal{A}$ by running $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$ and $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_1)$, storing the generated ballots in boards BB_0 and BB_1, and returning the ballot for v_1 to $\mathcal{A}$. When $\mathcal{A}$ produces a board BB, $\mathcal{C}$ computes $\mathsf{BB}' = \overline{\textsc{Recover}^{\mathsf{del}'}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})}(\mathsf{BB}_0)$, and returns this board.

When $\beta = 1$, $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},1}$ and $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV2},1}$ give $\mathcal{A}$ the same view, and hence return the same result, except if $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')) \neq r$, where $(r, \Pi) = \mathsf{Tally}(\mathsf{BB}', \mathsf{sk})$. That is, except if $\mathsf{Exp}_{\mathcal{C},\mathcal{V}}^{\mathsf{SC}}$ returns 1. Therefore

$$|\mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV1},1}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\text{mb-BPRIV2},1}(\lambda) = 1)| \leq \mathbb{P}(\mathsf{Exp}_{\mathcal{C},\mathcal{V}}^{\mathsf{SC}}(\lambda) = 1).$$

Therefore:

$$\begin{aligned} & |\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda)=1)-\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda)=1)| \\ \leq\ & |\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda)=1)-\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda)=1)| \\ & +|\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda)=1)-\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda)=1)| \\ & +|\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda)=1)-\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda)=1)| \\ \leq\ & |\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda)=1)-\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda)=1)| \\ & +\mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{B},\mathcal{V}}(\lambda)=1)+\mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}(\lambda)=1), \end{aligned}$$

which implies, by the strong consistency assumption, that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\text{mb-BPRIV2}}$.

$\mathsf{Exp}^{\text{mb-BPRIV2}} \wedge$ different random oracles $\Rightarrow \mathsf{Exp}^{\text{mb-BPRIV3}}$.

We now show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\text{mb-BPRIV3}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\text{mb-BPRIV3}}$. Consider an adversary $\mathcal{B}$ against $\mathsf{Exp}^{\text{mb-BPRIV2}}$, that runs $\mathcal{A}$ internally. During the voting and verification phases, $\mathcal{B}$ answers $\mathcal{A}$'s oracle queries by making the same calls to its oracles, once $\mathcal{A}$ returns a board BB, $\mathcal{B}$ returns this same board. $\mathcal{B}$ also runs its own random oracle, that is made available to $\mathcal{A}$, to simulate the random oracle used for the proof of correct tallying that $\mathcal{A}$ expects to receive. This is made possible by the assumption that this random oracle is not used in any other part of the protocol. Once $\mathcal{A}$ calls $\mathcal{O}\mathsf{tally}''$, $\mathcal{B}$ calls $\mathcal{O}\mathsf{tally}''$ and receives a result r. Manipulating the random oracle it runs, $\mathcal{B}$ then produces a proof $\Pi = \mathsf{SimProof}(r, \mathsf{BB})$, and returns (r, Π) to $\mathcal{A}$. When $\mathcal{A}$ makes its guess regarding β, $\mathcal{B}$ returns the same guess.

It is clear that $\mathcal{A}$ as run by $\mathcal{B}$ has the same view it would have in game $\mathsf{Exp}^{\text{mb-BPRIV3},\beta}_{\mathcal{A},\mathcal{V}}$. Hence

$$\begin{aligned} & |\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV3},0}_{\mathcal{A},\mathcal{V}}(\lambda)=1)-\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda)=1)| = \\ & |\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV2},0}_{\mathcal{B},\mathcal{V}}(\lambda)=1)-\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV2},1}_{\mathcal{B},\mathcal{V}}(\lambda)=1)| \end{aligned}$$

is negligible, which proves no adversary has a non-negligible advantage in $\mathsf{Exp}^{\text{mb-BPRIV3}}$.

$\mathsf{Exp}^{\text{mb-BPRIV3}} \wedge \mathsf{Exp}^{\mathsf{ZK}} \Rightarrow \mathsf{Exp}^{\text{mb-BPRIV}}$.

Finally we show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\text{mb-BPRIV}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\text{mb-BPRIV}}$. Consider $\mathcal{A}$ playing $\mathsf{Exp}^{\text{mb-BPRIV3}}$ instead.

It is clear that, when $\beta = 1$, $\mathsf{Exp}^{\text{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\text{mb-BPRIV},1}_{\mathcal{A},\mathcal{V}}$ are identical. Hence

$$\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV},1}_{\mathcal{A},\mathcal{V}}=1)=\mathbb{P}(\mathsf{Exp}^{\text{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}=1).$$

When $\beta = 0$, the outputs of $\mathsf{Exp}^{\text{mb-BPRIV},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\text{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}$ can only differ if $\mathcal{A}$ is able to distinguish the real proof of correct tallying Π returned by $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$ from the simulated proof $\mathsf{SimProof}(r, \mathsf{BB})$.

More precisely, let $\mathcal{B}$ be an adversary against $\mathsf{Exp}^{\mathsf{ZK},\beta'}_{\mathcal{V}}$. $\mathcal{B}$ runs $\mathsf{Exp}^{\text{mb-BPRIV},0}_{\mathcal{A},\mathcal{V}}$ internally. That is, $\mathcal{B}$ generates its own sets of credentials U, using the $\mathsf{Register}$ algorithm, and runs $\mathcal{A}$, answering calls to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ by computing $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$. $\mathcal{B}$ obtains a board BB from $\mathcal{A}$, on which it performs the validity check using $\mathsf{ValidBoard}$. If this check fails, $\mathcal{B}$ answers 0. Otherwise, it lets $\mathcal{A}$ call its verification oracle, using Verify to answer $\mathcal{A}$'s queries. Again, if $\mathcal{A}$ does not make all voters in $\mathsf{H}_{\mathsf{check}}$ verify, $\mathcal{B}$ answers 0. If some verifications fail, $\mathcal{B}$ asks $\mathcal{A}$ for its guess

regarding the bit β', and returns it. Otherwise, $\mathcal{B}$ returns the board BB, and obtains (r, Π_β), where r is the result of the tally, and $\Pi_{\beta'}$, depending on β', is either the real proof Π_0 computed by $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$, or the simulated proof $\Pi_1 = \mathsf{SimProof}(r, \mathsf{BB})$. $\mathcal{B}$ continues to run $\mathcal{A}$, answering (r, Π_β) when $\mathcal{A}$ calls $\mathcal{O}\mathsf{tally}$. Finally $\mathcal{A}$ makes a guess regarding β', that $\mathcal{B}$ returns as its output.

It is clear that, if $\beta' = 0$, $\mathcal{A}$ as simulated by $\mathcal{B}$ in $\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}$ has the same view it would have in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},0}_{\mathcal{A},\mathcal{V}}$. Hence, $\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},0}_{\mathcal{A},\mathcal{V}} = 1)$.

Similarly if $\beta' = 1$, $\mathcal{A}$ simulated by $\mathcal{B}$ in $\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}$ has the same view it would have in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}}$. Hence, $\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}} = 1)$.

Thus,

$$\begin{aligned}
& |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
= \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
\leq \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
= \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|,
\end{aligned}$$

which implies that, assuming that the zero-knowledge property holds, no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$. □

B.3.5 Ideal functionality corresponding to Recover$^{\mathsf{del}'}$

Recall the predicate P^{del} that is checked by the functionality $\mathcal{F}^{\mathsf{del}}_{\mathsf{v}}(\rho)$:

$P^{\mathsf{del}}(L, f) = \top$ iff:

- f keeps the votes of all voters who check, in the same order for each voter:
  $\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i \in [\![1, |L|]\!] | \exists v.\ L[i] = (id, v)] =$
  $[f(j), j = 1 \ldots |\mathrm{dom}(f)| | \exists v.\ L[f(j)] = (id, v)]$
- and no honest votes are modified by f:
  $\forall i, id, v.\ f(i) = (id, v) \implies id \in \mathsf{D}$.

Theorem B.17. *The* Recover$^{\mathsf{del}'}$ *algorithm defined above is compatible with the predicate* P^{del}.

Proof. Indeed, consider an adversary $\mathcal{A}$, that plays the game $\mathsf{Exp}^{\mathsf{comp},P^{\mathsf{del}},\textsc{Recover}^{\mathsf{del}'}}_{\mathcal{A},\mathcal{V}}$. Following this game, let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, $\mathsf{U} \leftarrow \mathsf{Register}(1^\lambda, \mathcal{I})$, $\mathsf{CU} \leftarrow [\mathsf{U}[id] | id \in \mathsf{D}]$, and $\mathsf{PU} \leftarrow [\mathsf{Pub}(\mathsf{U}[id]) | id \in \mathcal{I}]$. $\mathcal{A}$ has access to the $\mathcal{O}\mathsf{vote}$ oracle, to generate honest ballots, that get stored in a board BB_1. For each $(id, (p, b)) \in \mathsf{BB}_1$, by construction, (p, b) was constructed by calling $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$ for some v. Let BB be the board returned by $\mathcal{A}$ in the game. Note that, by the non-malleability assumption, following the same reasoning as in the previous proof, BB_1 contains no duplicate ballots, except with negligible probability. If BB is not valid, or BB_1 does not contain at least one entry for each $id \in \mathsf{H}_{\mathsf{check}}$, $\mathcal{A}$ loses the game.

Otherwise let $\pi = \textsc{Recover}^{\mathsf{del}'}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$. Let us show that π is compatible with P^{del} *w.r.t.* $\mathsf{sk}, \mathsf{U}, \mathsf{L}_{id}$, except with negligible probability. Assume BB_1 contains no duplicate ballots, which, as explained, holds with overwhelming probability.

Let L be a list of elements (id, v), such that $[id | (id, v) \in L] = \mathsf{L}_{id}$. We show that $P^{\mathsf{del}}(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)) = \top$.

Consider the lists LL, LL', LL_{cast}, I, constructed by the following process:

$LL \leftarrow []; LL_{\mathsf{cast}} \leftarrow []; I \leftarrow []$;
$\mathsf{BB}_{\mathsf{revote}} \leftarrow \mathsf{BB}$ where only the last (p, b) is kept for each public key p
for $(p, b) \in \mathsf{BB}_{\mathsf{revote}}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $LL \leftarrow LL \parallel j; I[id] \leftarrow j$;
    (by assumption on BB_1, this j is unique)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ **then**
    $LL \leftarrow LL \parallel \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b); LL_{\mathsf{cast}} \leftarrow LL_{\mathsf{cast}} \parallel \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b)$;
(note that, since BB is valid, each distinct p in BB is associated with a different id,
and since each p occurs only once in $\mathsf{BB}_{\mathsf{revote}}$, $I[id]$ only gets modified once for each id,
and occurs only once in LL.)
$\forall id \in \mathsf{H}_{\mathsf{check}}.\ LL_{id} \leftarrow [i | \exists p, b.\ \mathsf{BB}_1[i] = (id, (p, b))]$
if $\forall id \in \mathsf{H}_{\mathsf{check}}.\ I[id] =$ the last element of LL_{id} **then**
  $LL' \leftarrow LL$
  for $id \in \mathsf{H}_{\mathsf{check}}$ **do**
    insert LL_{id} without its last element right before $I[id]$ in LL'
  return $(\lambda i.\ LL'[i])$
else
  $LL' \leftarrow [1 \dots |\mathsf{BB}_1|] \parallel LL_{\mathsf{cast}}$;
  return $(\lambda i.\ LL'[i])$

By definition, $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$ is $\lambda i. LL''[i]$, where LL'' is the list obtained obtained by removing all $\perp$ elements from LL'.

We have to show that $P^{\mathsf{del}}(L, \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)) = \top$. That is, that

1. No honest votes are modified by $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$:

$$\forall i.\ \forall (id, v).\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)[i] = (id, v) \implies id \in \mathsf{D}.$$

   Let i, id, v be such that $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)[i] = (id, v)$. Hence $LL'[j] = (id, v)$ for some j. Thus, by construction of LL', (id, v) is an element of LL_{cast}, which means that $\mathsf{BB}[k] = (p, b)$ for some k, p, b such that (p, b) does not occur in BB_1, $\mathsf{extract}_{\mathsf{id}}(p, \mathsf{U}) \notin \mathsf{H}$, and $\mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) = (id, v)$. Hence $id \in \mathsf{D}$.

2. $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$ keeps the votes of all voters who check, in the same order:

$$\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i \in [\![1, |L|]\!] | \exists v.\ L[i] = (id, v)] = [\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)(j) | \exists v.\ L[\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)(j)] = (id, v)],$$

   that is, since the non-$\perp$ elements in LL'' and LL' are in the same order,

$$\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i \in [\![1, |L|]\!] | \exists v.\ L[i] = (id, v)] = [LL'[j] | \exists (p, b).\ \mathsf{BB}_1[LL'[j]] = (id, (p, b))],$$

   i.e.

$$\forall id \in \mathsf{H}_{\mathsf{check}}.\ LL_{id} = [LL'[j] | \exists (p, b).\ \mathsf{BB}_1[LL'[j]] = (id, (p, b))].$$

   Let $id \in \mathsf{H}_{\mathsf{check}}$. We distinguish two cases:

- either the test $\forall id \in \mathsf{H}_{\mathsf{check}}.\ I[id] =$ the last element of LL_{id} succeeds: in that case, LL' is by definition obtained by adding all elements of LL_{id} except the last one before the only occurrence of this last element, $I[id]$, in LL. In addition, by definition, all p in $\mathsf{BB}_{\mathsf{revote}}$ are distinct, and since BB is valid, they are associated with different id's. Thus, by construction of LL, two distinct indices corresponding to ballots of the same id in BB_1 cannot occur in LL. Therefore, no element from LL_{id} is already present in LL, except for the last one $I[id]$. Hence, the list $[LL'[j] | \exists (p,b).\ \mathsf{BB}_1[LL[j]] = (id,(p,b))]$ contains exactly the elements of LL_{id}, in the same order, which proves the claim in this case.
- or this test fails, and $LL' = [1 \dots |\mathsf{BB}_1|] \parallel LL_{\mathsf{cast}}$. In that case it is clear that $[LL'[j] | \exists (p,b).\ \mathsf{BB}_1[LL'[j]] = (id,(p,b))] = [j | \exists (p,b).\ \mathsf{BB}_1[j] = (id,(p,b))]$.

In any case, the claim holds, which concludes the proof.

□

B.3.6 Conclusion on Belenios with revote

Theorem B.18. *Assume Belenios, with revote allowed, is strongly consistent, has non-malleable ballots and credentials, that the proofs of correct tallying are zero-knowledge, that different random oracles are used for these proofs and the rest of the protocol, and that ρ encodes the "last vote counts" revote policy.*

Then Belenios securely implements the three ideal functionalities $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$.

Proof.

- It directly follows from Theorems 11, B.16, and B.17 that, under these assumptions, Belenios securely implements the functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$.
- It is also clear that $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ is stronger than $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$, in that it gives less power to the simulator. Indeed these two functionalities are the same, except that $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ allows more modification functions from $\mathcal{S}$ than $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$. Any simulator $\mathcal{S}$ running with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ can thus be accurately simulated by a simulator $\mathcal{S}'$ running with $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$, that runs $\mathcal{S}$ internally, checks whether its proposed modification function f is accepted by P^{del}, then submits f if so, and blocks the result otherwise. (Note $\mathcal{S}'$ can check this condition using only f and the sequence of the ids of voters who submitted votes, but does not need to know the votes themselves, by definition of P^{del}). Therefore, any voting scheme that securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ also securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$, which proves the claim that Belenios implements this functionality.
- With a similar reasoning, it appears that $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ is also stronger than $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$, which implies that Belenios also securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$.

□

B.4 Case study: Civitas with revote

We show here that, under reasonable assumptions, Civitas with revote is mb-BPRIV *w.r.t.* $\textsc{Recover}^{\mathsf{del}}$ (described below), and that this implies it realises $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}$. Consequently it also realises the weaker $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$.

B.4.1 Notations

Compared to the "no revote" case, the verification algorithm differs. $\mathsf{Verify}(id, s, \mathsf{BB})$ now checks whether the last ballot recorded in s, *i.e.* the last ballot generated by id, appears on BB.

Since revote is now allowed, the validity algorithm $\mathsf{ValidBoard}$ also changes: it no longer rejects a board that contains several ballots associated with public credentials encrypting the same private credential. However it still checks all the proofs and performs the PETs on all ballots.

B.4.2 Recovery

We consider the following recovery algorithm:

$\textsc{Recover}^{\mathsf{del}}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$

$L \leftarrow []; L_{\mathsf{cast}} \leftarrow []; \forall id \in \mathsf{H}.\ L_{id} \leftarrow [];$
for $(p, b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id, (p, b))$ **then**
    $L \leftarrow L \parallel j; L_{id} \leftarrow L_{id} \parallel j;$
    (in case several such j exist, pick the first one)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ **then**
    $L \leftarrow L \parallel (p, b); L_{\mathsf{cast}} \leftarrow L_{\mathsf{cast}} \parallel (p, b);$
if $\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i | \exists p, b.\ \mathsf{BB}_1[i] = (id, (p, b))] = L_{id}$ **then**
  return $(\lambda i.\ L[i])$
else
  $L' \leftarrow [1 \ldots |\mathsf{BB}_1|] \parallel L_{\mathsf{cast}};$
  return $(\lambda i.\ L'[i])$

B.4.3 Assumptions

We assume, as before, that the ballots as well as the credentials are non malleable, *i.e.* that for all adversary $\mathcal{A}$,

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|$$

and $|\mathbb{P}(\mathsf{Exp}^{\mathsf{CNM}}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|$ are negligible in λ.

As before, we also assume that the voting scheme is strongly consistent, that the proof of correct tallying has the zero-knowledge property, and that the proof of correct tallying and the rest of the protocol use different random oracles.

In addition, as explained in Section 6.6.1, when a voter revotes in Civitas, her new ballot must indicate which ballot it replaces, and to provide proofs of knowledge of the contents of this previous ballot. The $\mathsf{ValidBoard}$ algorithm is in charge of checking all these proofs, and that the old ballot that is claimed to be replaced is indeed present on the bulletin board before the new ballot, and is not already supposed to be replaced. This chains the ballots belonging to the same voter together, and provided that this verification is correctly performed, if one of the ballot created by a voter is present in a valid board, it must mean that all of her previous ballots are also present exactly once, in the correct order. We formalise the assumption that the $\mathsf{ValidBoard}$ algorithm correctly performs all these checks by the following game.

$\mathsf{Exp}^{\mathsf{valid}}_{\mathcal{A},\mathcal{V}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $c \leftarrow \mathsf{Register}(1^\lambda, id); \mathsf{U}[id] \leftarrow c; \mathsf{PU}[id] \leftarrow \mathsf{Pub}(c)$
for all $id \in \mathsf{D}$ **do** $\mathsf{CU}[id] \leftarrow \mathsf{U}[id]$
$\mathsf{BB} \leftarrow \mathcal{A}^{\mathcal{O}\mathsf{vote}}(\mathsf{pk},\mathsf{CU},\mathsf{PU})$
if $\mathsf{ValidBoard}(\mathsf{BB},\mathsf{pk}) \wedge$
  $\exists id \in \mathsf{H}.\ \exists i,j,p,b.\ \mathsf{BB}[i] = (p,b) \ \wedge\ \mathsf{L}[id][j] = (p,b) \wedge$
  $[(p',b') \in \mathsf{BB}[1\ldots i] | (p',b') \in \mathsf{L}[id][1\ldots j]] \neq \mathsf{L}[id][1\ldots j]$
then return 1
else return 0.

$\mathcal{O}\mathsf{vote}(id, v)$

$(p, b, state) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$
$\mathsf{L}[id] \leftarrow \mathsf{L}[id] \parallel (p, b)$
return (p, b).

B.4.4 Civitas with revote is mb-BPRIV

Theorem B.19. *Assume Civitas, with revote allowed, is strongly consistent, has non-malleable ballots and credentials, that the proofs of correct tallying are zero-knowledge, that different random oracles are used for these proofs and the rest of the protocol, and that* ValidBoard *correctly checks the chain of ballots belonging to the same voters.*

Then Civitas is mb-BPRIV w.r.t. $\textsc{Recover}^{\mathsf{del}}$.

Proof. As for the "no revote" case, we will consider a sequence of games that differ from $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover}^{\mathsf{del}}}$ by the way the result the adversary gets to see is computed. These games are the same as before, except they use $\textsc{Recover}^{\mathsf{del}}$ instead of $\textsc{Recover}^{\mathsf{del,reorder}}$:

- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ is not shown the tally of the board/the recovered board by $\mathcal{O}\mathsf{tally}$, $\mathcal{O}\mathsf{tally}'$ instead computes the result using ρ and the extract function, without any proof of correct tallying.
- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ has access to $\mathcal{O}\mathsf{tally}''$, who computes the result of the election using the Tally algorithm, but still no proof of correct tallying.
- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ is shown by $\mathcal{O}\mathsf{tally}'''$ the result of the election computed by the Tally algorithm, but the proof of correct tallying for the board provided by $\mathcal{A}$ is simulated on both sides.

The structure of the proof is similar to the "no revote" case. We first show that if no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{NM}}$, $\mathsf{Exp}^{\mathsf{CNM}}$ or $\mathsf{Exp}^{\mathsf{valid}}$, then no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$. Using the strong consistency assumption, we then show that this implies no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. We then show, using the assumption that the random oracles used for the proof of correct tallying and the remainder of the protocol are different, that this implies no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. From there, using the zero-knowledge assumption on the proof of correct tallying, we show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$.

$\mathsf{Exp}^{\mathsf{NM}} \ \wedge\ \mathsf{Exp}^{\mathsf{CNM}} \ \wedge\ \mathsf{Exp}^{\mathsf{valid}} \Rightarrow \mathsf{Exp}^{\mathbf{mb\text{-}BPRIV1}}$.

Let $\mathcal{A}$ be an attacker that wins the $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},}_{\mathcal{V}}$ game.

We construct an attacker $\mathcal{B}$ against $\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{V}}$. $\mathcal{B}$ is given access to pk and U. It runs $\mathcal{A}$ internally, simulating the oracle calls as follows. When $\mathcal{A}$ calls $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$, for some

$id \in \mathsf{H}$, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{c}}(id, \mathsf{U}[id], v_0, v_1)$ and obtains some (p, b). $\mathcal{B}$ records $(id, (p, b))$ in a list $\mathsf{BB}_{\mathsf{init}}$, and stores (id, v_0, v_1, p, b) in a list V. $\mathcal{B}$ then returns (p, b) to $\mathcal{A}$. At some point, $\mathcal{A}$ returns to $\mathcal{B}$ some board BB.

Note that

- By construction, during the execution, the list $[(p, b) | (id, (p, b)) \in \mathsf{BB}_{\mathsf{init}}]$ of the ballots in $\mathsf{BB}_{\mathsf{init}}$ (kept by $\mathcal{B}$) is always equal to the list L in the game $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{NM},\beta}$ played by $\mathcal{B}$.
- It is also clear by examining the game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{mb\text{-}BPRIV1},\beta}$ that, up to this point, $\mathcal{A}$ has been accurately simulated, and has the same view it would have in game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{mb\text{-}BPRIV1},\beta}$.
- By construction of the $\mathcal{O}\mathsf{voteLR}$ oracle, the list $\mathsf{BB}_{\mathsf{init}}$ is also equal to the list BB_β in (the corresponding execution of) the game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{mb\text{-}BPRIV1},\beta}$ for $\mathcal{A}$.
- We can also note that $\mathsf{BB}_{\mathsf{init}}$ contains no duplicate ballots (except with a negligible probability $p_1^\beta(\lambda)$). Indeed, $\mathsf{BB}_{\mathsf{init}}$ containing duplicate ballots would imply that two oracle calls $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v)$ and $\mathsf{Vote}(\mathsf{pk}, id', \mathsf{U}[id'], v')$ produced the same ballot. If this happened with non-negligible probability, by submitting $\mathcal{O}_{\mathsf{c}}(id, \mathsf{U}[id], v, v'')$ followed by $\mathcal{O}_{\mathsf{c}}(id', \mathsf{U}[id'], v', v''')$ (for some $v'' \neq v'''$) to the encryption oracle, and checking whether the two resulting ballots are equal, an adversary would win the non-malleability game. The two ballots obtained on the right would indeed be different, except with negligible probability, by strong consistency (as they contain different votes).

Once $\mathcal{A}$ has returned BB, $\mathcal{B}$ checks (in V) that all voters in $\mathsf{H}_{\mathsf{check}}$ have voted, and halts if not. $\mathcal{B}$ also checks whether $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$. If not, $\mathcal{B}$ directly asks $\mathcal{A}$ to guess the bit β', and returns $\mathcal{A}$'s guess.

Following the structure of game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$, $\mathcal{B}$ then lets $\mathcal{A}$ perform the verifications for the honest voters. That is, when $\mathcal{A}$ calls $\mathcal{O}\mathsf{verify}_{\mathsf{BB}}(id)$, $\mathcal{B}$ retrieves the last entry (id, v_0, v_1, p, b) for id in V, and checks that $(p, b) \in \mathsf{BB}$. Once $\mathcal{A}$ is done with the verification phase, $\mathcal{B}$ checks that each $id \in \mathsf{H}_{\mathsf{check}}$ has verified (and halts otherwise). If any of the verifications have failed, $\mathcal{B}$ directly asks $\mathcal{A}$ to guess the bit β', and returns $\mathcal{A}$'s guess.

If all verifications are successful, $\mathcal{B}$ will simulate the tallying phase to $\mathcal{A}$. To do this, $\mathcal{B}$ first computes $\pi = \textsc{Recover}_{\mathsf{U}}^{\mathsf{del}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB})$.

Note, at this point, that since all checks succeeded, π is simpler than in the general case. Indeed, the $\mathsf{Exp}^{\mathsf{valid}}$ assumption implies that, except with negligible probability $p_2^\beta(\lambda)$, for each voter $id \in \mathsf{H}_{\mathsf{check}}$, all the ballots generated for id are present in BB in the same order. Otherwise, an adversary could win the $\mathsf{Exp}^{\mathsf{valid}}$ game by running $\mathcal{A}$ and returning the same board BB. Therefore, the test "**if** $\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i | \exists p, b.\ \mathsf{BB}_{\mathsf{init}}[i] = (id, (p, b))] = L_{id}$" performed by $\textsc{Recover}_{\mathsf{U}}^{\mathsf{del}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB})$ succeeds.

Hence, π is only defined on $[\![1, |\mathsf{BB}|]\!]$, and, except with negligible probability $p_3^\beta(\lambda)$, is such that for all i:

- $\pi(i) = j$ if j is the index of the first (and only) occurrence of $\mathsf{BB}[i]$ in $\mathsf{BB}_{\mathsf{init}}$,
- or $\pi(i) = \mathsf{BB}[i]$ if no such index exists. Indeed, in that case, we have $\mathsf{extract}_{\mathsf{id}}(\mathsf{U}, p) \notin \mathsf{H}$ (where p is the public credential in $\mathsf{BB}[i]$), except with negligible probability: otherwise an adversary could win game $\mathsf{Exp}^{\mathsf{CNM}}$ by simulating $\mathcal{A}$ and returning BB.

To continue the simulation of $\mathcal{A}$, $\mathcal{B}$ must then provide $\mathcal{A}$ with the result of the election, to answer $\mathcal{A}$'s calls to $\mathcal{O}\mathsf{tally}'$. $\mathcal{B}$ asks for the decryption of the list of all (p,b) such that $\exists j.\ \pi(j) = (p,b)$. That is, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{d}}([(p,b)|\exists j.\ \pi(j) = (p,b)])$. This call to the decryption oracle is allowed: indeed, by definition of $\textsc{Recover}^{\mathsf{del}}$, a ballot such that $\pi(j) = (p,b)$ cannot occur in an element of $\mathsf{BB}_{\mathsf{init}}$, and hence is not in L.

$\mathcal{B}$ then constructs a list $\overline{\mathsf{BB}}$ containing the votes in $\overline{\pi}\mathsf{BB}_0$ in clear, BB_0 being the board of the left ballots, maintained by $\mathcal{O}\mathsf{voteLR}$ in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$. This is done by retrieving the vote from the list V for ballots coming from the encryption oracle, and using the decryption oracle for the others. Formally this list is obtained by, for all i:

- $\overline{\mathsf{BB}}[i] = (id, v_0)$ if $\mathsf{BB}[i] = (p,b)$ for some (p,b) appearing in $\mathsf{BB}_{\mathsf{init}}$, where (id, v_0, v_1, p, b) is the corresponding element in V (in that case, $\pi(i)$ is the index of this element in V).
- $\overline{\mathsf{BB}}[i] = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}[i])$, which $\mathcal{B}$ gets from its call to $\mathcal{O}_{\mathsf{d}}$, if $\mathsf{BB}[i]$ does not appear in $\mathsf{BB}_{\mathsf{init}}$.

At this point, we have $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))$, except with negligible probability. Indeed, for any i:

- Either $\mathsf{BB}[i]$ does not appear in $\mathsf{BB}_{\mathsf{init}}$, and, by definition, $\pi(i) = \mathsf{BB}[i]$. Hence $\overline{\pi}(\mathsf{BB}_0)[i] = \mathsf{BB}[i]$. Then $\overline{\mathsf{BB}}[i] = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}[i])$ holds by construction of $\overline{\mathsf{BB}}$.
- Or $\mathsf{BB}_{\mathsf{init}}[j] = (id, \mathsf{BB}[i])$ for some j, id, and, by definition, $\pi(i) = j$. Hence $\overline{\pi}(\mathsf{BB}_0)[i] = \mathsf{BB}[i]$. Let $(id, v_0, v_1, \mathsf{BB}[i])$ denote $\mathsf{V}[j]$. By construction by the $\mathcal{O}\mathsf{voteLR}$ oracle, $\mathsf{BB}_0[j] = (id, (p,b))$, where (p,b) is a ballot created by calling $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$. By construction of $\overline{\mathsf{BB}}$, $\overline{\mathsf{BB}}[i] = (id, v_0)$. Hence,

$$\begin{array}{rl} & \mathbb{P}(\overline{\mathsf{BB}}[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}[i]))) \\ \leq & \mathbb{P}((\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda);\ \mathsf{U} \leftarrow \mathsf{Register}(1^\lambda, \mathcal{I}); \\ & \quad (s, p, b) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0);\ \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) \neq (id, v_0)) \end{array}$$

  which is negligible by strong consistency.

Since $\overline{\mathsf{BB}}$ has as many elements as BB, it is of polynomial size (bounded by the running time of $\mathcal{A}$). Hence, $\mathbb{P}(\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))) = \mathbb{P}(\exists i.\ \overline{\mathsf{BB}}[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0)[i]))$ is also negligible. We write

$$p_4^\beta(\lambda) = \mathbb{P}(\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0)) \text{ in } \mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{B},\mathcal{V}}).$$

$\mathcal{B}$ then computes $\rho(\overline{\mathsf{BB}})$, and shows this result to $\mathcal{A}$. $\mathcal{A}$ finally outputs a bit β', that $\mathcal{B}$ returns.

We now argue that $\rho(\overline{\mathsf{BB}})$ is indeed the result $\mathcal{A}$ would have been shown in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$.

- If $\beta = 1$, $\mathsf{BB}_{\mathsf{init}} = \mathsf{BB}_1$, and thus $\pi = \textsc{Recover}^{\mathsf{del}}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$. As previously explained, we then have

$$\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\textsc{Recover}^{\mathsf{del}}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})}(\mathsf{BB}_0)),$$

  that is, $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$, where BB' is the board computed using $\textsc{Recover}^{\mathsf{del}}$ in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}$. By definition of $\mathcal{O}\mathsf{tally}'$, $\rho(\overline{\mathsf{BB}})$ is thus the election result $\mathcal{A}$ sees in this game.

- If $\beta = 0$, $\mathsf{BB}_{\mathsf{init}} = \mathsf{BB}_0$, and thus $\pi = \textsc{Recover}_{\mathsf{U}}^{\mathsf{del}}(\mathsf{BB}_0, \mathsf{BB})$. Recall that, since all the verifications have succeeded, for each honest voter who checks $id \in \mathsf{H}_{\mathsf{check}}$, all the ballots in BB_0 produced by id occur in BB in the same order (except with negligible probability $p_2^\beta(\lambda)$). Therefore, it is clear from the definition of $\textsc{Recover}^{\mathsf{del}}$ that $\overline{\pi}(\mathsf{BB}_0) = \mathsf{BB}$. As previously explained, we then have $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})$, and, by definition of $\mathcal{O}\mathsf{tally}'$, $\rho(\overline{\mathsf{BB}})$ is thus the election result $\mathcal{A}$ sees in this $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}$.

As we argued, except if

- $\mathsf{BB}_{\mathsf{init}}$ contains duplicate ballots ($p_1^\beta(\lambda)$),
- or the validity check fails to ensure all previous ballots of each voter who check are there ($p_2^\beta(\lambda)$),
- or π does not have the expected form ($p_3^\beta(\lambda)$),
- or $\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}\mathsf{BB}_0)$ ($p_4^\beta(\lambda)$),
- or applying π to BB_0 does not return BB when $\beta = 0$ ($p_2^\beta(\lambda)$),

$\mathcal{A}$ is run until the end of its execution by $\mathcal{B}$ if and only if it would also reach the end of the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A}}$; and $\mathcal{A}$ run by $\mathcal{B}$ has the same view it would have in the corresponding execution of $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$. Hence, except in those five cases, $\mathcal{B}$ returns 1 in $\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{B}}$ if and only if $\mathcal{A}$ returns 1 in the corresponding execution of $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A}}$. Thus, for any β, if $p^\beta(\lambda)$ denotes $p_1^\beta(\lambda) + 2 * p_2^\beta(\lambda) + p_3^\beta(\lambda) + p_4^\beta(\lambda)$,

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \leq p^\beta(\lambda).$$

Therefore

$$\begin{array}{rl} & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\ & + |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\ & + |\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| + p^0(\lambda) + p^1(\lambda), \end{array}$$

which implies that, assuming strong consistency holds, and that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{NM}}$, no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$.

The remaining steps of the proof are identical to the "no revote" case.

$\underline{\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}} \wedge \mathsf{Exp}^{\mathsf{SC}} \Rightarrow \mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}.}$

We now show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. Consider the same $\mathcal{A}$ playing $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$ instead. $\mathcal{A}$ has the same view in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$ as in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},\beta}_{\mathcal{A},\mathcal{V}}$, except if the result r returned by the Tally algorithm differs from ρ applied to the extractions of the board, which happens only with negligible probability by the second point of the strong consistency assumption.

More precisely, consider an adversary $\mathcal{B}$ playing $\mathsf{Exp}^{\mathsf{SC}}$, that runs $\mathcal{A}$ internally, simulating its execution in game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}$. $\mathcal{B}$ answers any call to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ made by $\mathcal{A}$ by

running $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$, and returning the generated ballot to $\mathcal{A}$. When $\mathcal{A}$ produces a board BB, $\mathcal{B}$ returns this board.

When $\beta = 0$, $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}$ give $\mathcal{A}$ the same view, and hence return the same result, except if $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})) \neq r$, where $(r, \Pi) = \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$. That is, except if $\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{B},\mathcal{V}}$ returns 1. Therefore

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \leq \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{B},\mathcal{V}}(\lambda) = 1).$$

Similarly, consider an adversary $\mathcal{C}$ playing $\mathsf{Exp}^{\mathsf{SC}}$, that runs $\mathcal{A}$ internally, simulating its execution in game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}$. $\mathcal{C}$ answers any call to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ made by $\mathcal{A}$ by running $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$ and $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_1)$, storing the generated ballots in boards BB_0 and BB_1, and returning the ballot for v_1 to $\mathcal{A}$. When $\mathcal{A}$ produces a board BB, $\mathcal{C}$ computes $\mathsf{BB}' = \overline{\textsc{Recover}^{\mathsf{del}}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})}(\mathsf{BB}_0)$, and returns this board.

When $\beta = 1$, $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}$ give $\mathcal{A}$ the same view, and hence return the same result, except if $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')) \neq r$, where $(r, \Pi) = \mathsf{Tally}(\mathsf{BB}', \mathsf{sk})$. That is, except if $\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}$ returns 1. Therefore

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \leq \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}(\lambda) = 1).$$

Therefore:

$$\begin{array}{rl} & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & +|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & +|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & +\mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) + \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}(\lambda) = 1), \end{array}$$

which implies, by the strong consistency assumption, that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$.

$\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}} \wedge$ different random oracles $\Rightarrow \mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$.

We now show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. Consider an adversary $\mathcal{B}$ against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$, that runs $\mathcal{A}$ internally. During the voting and verification phases, $\mathcal{B}$ answers $\mathcal{A}$'s oracle queries by making the same calls to its oracles, once $\mathcal{A}$ returns a board BB, $\mathcal{B}$ returns this same board. $\mathcal{B}$ also runs its own random oracle, that is made available to $\mathcal{A}$, to simulate the random oracle used for the proof of correct tallying that $\mathcal{A}$ expects to receive. This is made possible by the assumption that this random oracle is not used in any other part of the protocol. Once $\mathcal{A}$ calls $\mathcal{O}\mathsf{tally}'''$, $\mathcal{B}$ calls $\mathcal{O}\mathsf{tally}''$ and receives a result r. Manipulating the random oracle it runs, $\mathcal{B}$ then produces a proof $\Pi = \mathsf{SimProof}(r, \mathsf{BB})$, and returns (r, Π) to $\mathcal{A}$. When $\mathcal{A}$ makes its guess regarding β, $\mathcal{B}$ returns the same guess.

It is clear that $\mathcal{A}$ as run by $\mathcal{B}$ has the same view it would have in game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},\beta}_{\mathcal{A},\mathcal{V}}$. Hence

$$\begin{array}{l} |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| = \\ \quad |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \end{array}$$

is negligible, which proves no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$.

$\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}} \wedge \mathsf{Exp}^{\mathsf{ZK}} \Rightarrow \mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$.

Finally we show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$. Consider $\mathcal{A}$ playing $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$ instead.

It is clear that, when $\beta = 1$, $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}}$ are identical. Hence

$$\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}} = 1).$$

When $\beta = 0$, the outputs of $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}$ can only differ if $\mathcal{A}$ is able to distinguish the real proof of correct tallying Π returned by $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$ from the simulated proof $\mathsf{SimProof}(r, \mathsf{BB})$.

More precisely, let $\mathcal{B}$ be an adversary against $\mathsf{Exp}^{\mathsf{ZK},\beta'}_{\mathcal{V}}$. $\mathcal{B}$ runs $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},0}_{\mathcal{A},\mathcal{V}}$ internally. That is, $\mathcal{B}$ generates its own sets of credentials U, using the $\mathsf{Register}$ algorithm, and runs $\mathcal{A}$, answering calls to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ by computing $\mathsf{Vote}(\mathsf{pk}, id, \mathsf{U}[id], v_0)$. $\mathcal{B}$ obtains a board BB from $\mathcal{A}$, on which it performs the validity check using $\mathsf{ValidBoard}$. If this check fails, $\mathcal{B}$ answers 0. Otherwise, it lets $\mathcal{A}$ call its verification oracle, using Verify to answer $\mathcal{A}$'s queries. Again, if $\mathcal{A}$ does not make all voters in $\mathsf{H}_{\mathsf{check}}$ verify, $\mathcal{B}$ answers 0. If some verifications fail, $\mathcal{B}$ asks $\mathcal{A}$ for its guess regarding the bit β', and returns it. Otherwise, $\mathcal{B}$ returns the board BB, and obtains (r, Π_β), where r is the result of the tally, and $\Pi_{\beta'}$, depending on β', is either the real proof Π_0 computed by $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$, or the simulated proof $\Pi_1 = \mathsf{SimProof}(r, \mathsf{BB})$. $\mathcal{B}$ continues to run $\mathcal{A}$, answering (r, Π_β) when $\mathcal{A}$ calls $\mathcal{O}\mathsf{tally}$. Finally $\mathcal{A}$ makes a guess regarding β', that $\mathcal{B}$ returns as its output.

It is clear that, if $\beta' = 0$, $\mathcal{A}$ as simulated by $\mathcal{B}$ in $\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}$ has the same view it would have in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},0}_{\mathcal{A},\mathcal{V}}$. Hence, $\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},0}_{\mathcal{A},\mathcal{V}} = 1)$.

Similarly if $\beta' = 1$, $\mathcal{A}$ simulated by $\mathcal{B}$ in $\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}$ has the same view it would have in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}}$. Hence, $\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}} = 1)$.

Thus,

$$\begin{aligned}
& |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
= \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
\leq \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
= \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|,
\end{aligned}$$

which implies that, assuming that the zero-knowledge property holds, no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$. □

B.4.5 Ideal functionality corresponding to Recover$^{\mathsf{del}}$

Recall the predicate P^{del} from the functionality $\mathcal{F}^{\mathsf{del}}_{\mathsf{v}}(\rho)$:

$P^{\mathsf{del}}(L, f) = \top$ iff:

- f keeps the votes of all voters who check, in the same order for each voter:
  $\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i \in [\![1, |L|]\!] | \exists v.\ L[i] = (id, v)] =$
  $[f(j), j = 1 \ldots |\mathrm{dom}(f)| | \exists v.\ L[f(j)] = (id, v)]$
- and no honest votes are modified by f:
  $\forall i, id, v.\ f(i) = (id, v) \implies id \in \mathsf{D}$.

Theorem B.20. *The* $\textsc{Recover}^{\mathsf{del}}$ *algorithm defined above is compatible with the predicate* P^{del}*.*

Proof. Indeed, consider an adversary $\mathcal{A}$, that plays the game $\mathsf{Exp}^{\mathsf{comp},P^{\mathsf{del}},\textsc{Recover}^{\mathsf{del}}}_{\mathcal{A},\mathcal{V}}$. Following this game, let $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, $\mathsf{U} \leftarrow \mathsf{Register}(1^\lambda,\mathcal{I})$, $\mathsf{CU} \leftarrow [\mathsf{U}[id]|id \in \mathsf{D}]$, and $\mathsf{PU} \leftarrow [\mathsf{Pub}(\mathsf{U}[id])|id \in \mathcal{I}]$. $\mathcal{A}$ has access to the $\mathcal{O}\mathsf{vote}$ oracle, to generate honest ballots, that get stored in a board BB_1. For each $(id,(p,b)) \in \mathsf{BB}_1$, by construction, (p,b) was constructed by calling $\mathsf{Vote}(\mathsf{pk},id,\mathsf{U}[id],v)$ for some v. Let BB be the board returned by $\mathcal{A}$ in the game. Note that, by the non-malleability assumption, following the same reasoning as in the previous proof, BB_1 contains no duplicate ballots, except with negligible probability. If BB is not valid, or BB_1 does not contain at least one entry for each $id \in \mathsf{H}_{\mathsf{check}}$, $\mathcal{A}$ loses the game.

Otherwise let $\pi = \textsc{Recover}^{\mathsf{del}}_{\mathsf{U}}(\mathsf{BB}_1,\mathsf{BB})$. Let us show that π is compatible with P^{del} *w.r.t.* $\mathsf{sk},\mathsf{U},L_{id}$, except with negligible probability. Assume BB_1 contains no duplicate ballots, which, as explained, holds with overwhelming probability.

Let L be a list of elements (id,v), such that $[id|(id,v) \in L] = \mathsf{L}_{id}$. We show that $P^{\mathsf{del}}(L,\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)) = \top$.

Consider the lists LL, LL', LL_{cast}, LL_{id} for $id \in \mathsf{H}$, constructed by the following process:

$LL \leftarrow []; LL_{\mathsf{cast}} \leftarrow []; \forall id \in \mathsf{H}.\ LL_{id} \leftarrow [];$
for $(p,b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id,(p,b))$ **then**
    $LL \leftarrow LL \parallel j; LL_{id} \leftarrow LL_{id} \parallel j;$
    (by assumption on BB_1, this j is unique)
  else if $\mathsf{extract}_{\mathsf{id}}(\mathsf{U},p) \notin \mathsf{H}$
    $LL \leftarrow LL \parallel \mathsf{extract}(\mathsf{sk},\mathsf{U},p,b); LL_{\mathsf{cast}} \leftarrow LL_{\mathsf{cast}} \parallel \mathsf{extract}(\mathsf{sk},\mathsf{U},p,b);$
if $\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i|\exists p,b.\ \mathsf{BB}_1[i] = (id,(p,b))] = LL_{id}$ **then**
  $LL' \leftarrow LL$
else $LL' \leftarrow [1 \ldots |\mathsf{BB}_1|] \parallel LL_{\mathsf{cast}};$

By definition, $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$ is $\lambda i.LL''[i]$, where LL'' is the list obtained obtained by removing all $\perp$ elements from LL'.

We have to show that $P^{\mathsf{del}}(L,\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)) = \top$. That is, that

1. No honest votes are modified by $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$:

$$\forall i.\ \forall (id,v).\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)[i] = (id,v) \implies id \in \mathsf{D}.$$

   Let i, id, v be such that $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)[i] = (id,v)$. Hence $LL'[j] = (id,v)$ for some j. Thus, by construction of LL', (id,v) is an element of LL_{cast}, which means that $\mathsf{BB}[k] = (p,b)$ for some k,p,b such that (p,b) does not occur in BB_1, $\mathsf{extract}_{\mathsf{id}}(p,\mathsf{U}) \notin \mathsf{H}$, and $\mathsf{extract}(\mathsf{sk},\mathsf{U},p,b) = (id,v)$. Hence $id \in \mathsf{D}$.

2. $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$ keeps the votes of all voters who check, in the same order:

$$\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i \in [\![1,|L|]\!]|\exists v.\ L[i] = (id,v)] = [\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)(j)|\exists v.\ L[\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)(j)] = (id,v)],$$

   that is, since the non-$\perp$ elements in LL'' and LL' are in the same order,

$$\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i|\exists (p,b).\ \mathsf{BB}_1[i] = (id,(p,b))] = [LL'[j]|\exists (p,b).\ \mathsf{BB}_1[LL'[j]] = (id,(p,b))].$$

   Let $id \in \mathsf{H}_{\mathsf{check}}$. We distinguish two cases:

- either the test $\forall id \in \mathsf{H}_{\mathsf{check}}.\ [i|\exists p, b.\ \mathsf{BB}_1[i] = (id, (p, b))] = LL_{id}$ succeeds: in that case, we have $LL' = LL$, and it is sufficient to show that $[LL[j]|\exists (p, b).\ \mathsf{BB}_1[LL[j]] = (id, (p, b))] = LL_{id}$, which is true by construction of LL_{id}.
- or this test fails, and $LL' = [1 \dots |\mathsf{BB}_1|] \parallel LL_{\mathsf{cast}}$. In that case it is clear that $[LL'[j]|\exists (p, b).\ \mathsf{BB}_1[LL'[j]] = (id, (p, b))] = [j|\exists (p, b).\ \mathsf{BB}_1[j] = (id, (p, b))]$.

In any case, the claim holds, which concludes the proof.

□

B.4.6 Conclusion on Civitas with revote

Theorem B.21. *Assume Civitas, with revote allowed, is strongly consistent, has non-malleable ballots and credentials, that the proofs of correct tallying are zero-knowledge, that different random oracles are used for these proofs and the rest of the protocol, and that* ValidBoard *correctly checks the chain of ballots belonging to the same voters.*

Then Civitas securely implements the three ideal functionalities $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ *and* $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$.

Proof.

- It directly follows from Theorems 11, B.19, and B.20 that, under these assumptions, Civitas securely implements the functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$.
- As argued for Belenios in Section B.3.6, $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ is stronger than both $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder}}(\rho)$ and $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}(\rho)$, meaning that any scheme that securely implements $\mathcal{F}_{\mathsf{v}}^{\mathsf{del}}(\rho)$ also implements these two functionalities, which proves the claim.

□

B.5 Case study: Helios without revote

We show here that, under reasonable assumptions, Helios without revote is mb-BPRIV *w.r.t.* $\textsc{Recover}^{\mathsf{del,reorder,change}'}$ (allowing only one call to the $\mathcal{O}$voteLR oracle for each id, where $\textsc{Recover}^{\mathsf{del,reorder,change}'}$ is defined below), and that this implies it realises $\mathcal{F}_{\mathsf{v}}^{\mathsf{del,reorder,change}}$ (when the environment $\mathcal{E}$ does not make voters revote).

B.5.1 Notations: Helios

- Helios does not use credentials, hence the Register and Pub algorithms are unused (they return empty strings).
- $\mathsf{Vote}(id, \mathsf{pk}, c, v) = (s, id, (\mathsf{enc}(v, \mathsf{pk}), \Pi))$. We will call $(\mathsf{enc}(v, \mathsf{pk}), \Pi)$ the ciphertext of the ballot. The state s records this ciphertext. The pseudonym is id. π is a zero-knowledge proof that v is a valid vote. Since Vote does not use a credential c, we will omit it in the following.
- $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk})$ checks the zero-knowledge proofs in the ballots in BB, that BB does not contain several ballots for the same id, and that BB does not contain duplicate ciphertexts.

- Tally only keeps the parts of the ballots containing the encrypted votes. These are then run through a mixnet, decrypted, and published.

- $\mathsf{Verify}(id, s, \mathsf{BB})$ checks whether the ciphertext recorded in s appears on BB.

B.5.2 Recovery

We consider the following recovery algorithm:

$\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$

$L \leftarrow []$
$S \leftarrow \mathsf{D} \cup \{id \in \mathsf{H}_{\overline{\mathsf{check}}} \mid \forall p, b.\ (id, (p, b)) \in \mathsf{BB}_1 \ \Rightarrow\ \forall p'.\ (p', b) \notin \mathsf{BB})\}$
for $(p, b) \in \mathsf{BB}$ **do**
  if $\exists j, id, p'.\ \mathsf{BB}_1[j] = (id, (p', b))$ **then**
    $L \leftarrow L \parallel j$
    (in case several such j exist, pick the first one)
  else if $S \neq \emptyset$ **then**
    $L \leftarrow L \parallel (id, b); S \leftarrow S \backslash \{id\}$
    (for some $id \in S$)
$L' \leftarrow [i | \mathsf{BB}_1[i] = (id, (p, b)) \ \wedge\ id \in \mathsf{H}_{\mathsf{check}} \ \wedge\ \forall p'.\ (p', b) \notin \mathsf{BB}]$
$L'' \leftarrow L \parallel L'$
return $(\lambda i.\ L''[i])$

Intuitively, given BB_1 and BB, when applied to BB_0, this recovery algorithm will construct a board BB' where

- all the ciphertexts in BB that come from BB_1 are replaced with the corresponding ciphertext from BB_0;

- the other ciphertexts in BB are considered to be cast, and added to BB' as is;

- all the ciphertexts registered for voters who check in BB_0 are added to BB', regardless of whether these voters' ballots actually occur in BB.

The subtle point is that the ciphertexts that are cast should be cast for identities that do not conflict with those appearing in BB_1: only the identities of dishonest voters, or of voters whose ballot was removed should be used. That is the purpose of the set S used by $\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}$.

B.5.3 Assumptions

We assume that the ciphertexts of the ballots (*i.e.* excluding the identity p) are non malleable, *i.e.* that for all adversary $\mathcal{A}$,

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0'}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1'}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|$$

is negligible in λ, where $\mathsf{Exp}^{\mathsf{NM},\beta'}_{\mathcal{A},\mathcal{V}}$ is the following game.

$\mathsf{Exp}^{\mathsf{NM},\beta'}_{\mathcal{A},\mathcal{V}}(\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$
for all $id \in \mathcal{I}$ **do**
  $\mathsf{U}[id] \leftarrow \mathsf{Register}(1^\lambda, id)$
$\beta' \leftarrow \mathcal{A}^{\mathcal{O}_\mathsf{c},\mathcal{O}_\mathsf{d}}(\mathsf{pk},\mathsf{U})$
output β'.

$\mathcal{O}_\mathsf{c}(id, v_0, v_1)$

$(p, b, state) \leftarrow \mathsf{Vote}(\mathsf{pk}, id, v_\beta)$
$\mathsf{L} \leftarrow \mathsf{L} \parallel b$
return b.

$\mathcal{O}_\mathsf{d}(\mathsf{cL})$

for all $b \in \mathsf{cL}$ **do**
  if $b \notin \mathsf{L}$ **then**
    $\mathsf{dL} \leftarrow \mathsf{dL} \parallel \mathsf{extract}_\mathsf{v}(\mathsf{sk}, b)$
return dL.

$\mathcal{A}$ is allowed any number of calls to $\mathcal{O}_\mathsf{c}$, followed by one single call to $\mathcal{O}_\mathsf{d}$.

As for Belenios and Civitas, we assume that the voting scheme is strongly consistent, and that the proof of correct tallying and the rest of the protocol use different random oracles. We also assume a zero-knowledge property from the proof of correct tallying, expressed, as before, by the game $\mathsf{Exp}^{\mathsf{ZK}}$.

In addition, we will assume that the counting function ρ only uses the identities it is given to apply a revote policy, that is, that it only looks for repetitions between the *id*s, but does not depend on the *id*s themselves. Formally we assume that for any list L of pairs (id, v), if σ is a permutation of identities, then $\rho(L) = \rho(L\sigma)$. We call this property *id*-blindness.

B.5.4 Helios without revote is mb-BPRIV

Theorem B.22. *Assume Helios, with no revote allowed, is strongly consistent, has non-malleable ciphertexts, that the proofs of correct tallying are zero-knowledge, that different random oracles are used for these proofs and the rest of the protocol, and that the counting function ρ is id-blind.*

Then Helios, without revote, is mb-BPRIV w.r.t. $\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}$.

Proof. As for Belenios and Civitas, we will consider a sequence of games that differ from $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV},\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}}$ by the way the result the adversary gets to see is computed. These games are the same as before, except they use the algorithm $\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}$ instead of $\textsc{Recover}^{\mathsf{del},\mathsf{reorder}}$:

- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ is not shown the tally of the board/the recovered board by $\mathcal{O}\mathsf{tally}$, $\mathcal{O}\mathsf{tally}'$ instead computes the result using ρ and the $\mathsf{extract}$ function, without any proof of correct tallying.
- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ has access to $\mathcal{O}\mathsf{tally}''$, who computes the result of the election using the Tally algorithm, but still no proof of correct tallying.
- In $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},\beta}_{\mathcal{A},\mathcal{V}}$, $\mathcal{A}$ is shown by $\mathcal{O}\mathsf{tally}'''$ the result of the election computed by the Tally algorithm, but the proof of correct tallying for the board provided by $\mathcal{A}$ is simulated on both sides.

The structure of the proof is similar to the case of Belenios and Civitas. We first show that if no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{NM}'}$, then no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$. Using the strong consistency assumption, we then show that this implies no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. We then show, using the assumption that the random oracles used for the proof of correct tallying and the remainder of the protocol are different, that this implies no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. From there, using the zero-knowledge assumption on the proof of correct tallying, we show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}}$.

$\underline{\mathsf{Exp}^{\mathsf{NM}'} \Rightarrow \mathsf{Exp}^{\textbf{mb-BPRIV}1}.}$

Let $\mathcal{A}$ be an attacker that wins the $\mathsf{Exp}_{\mathcal{V}}^{\text{mb-BPRIV}1,}$ game.

We construct an attacker $\mathcal{B}$ against $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{NM},\beta'}$. $\mathcal{B}$ is given access to pk and U. It runs $\mathcal{A}$ internally, simulating the oracle calls as follows. When $\mathcal{A}$ calls $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$, for some $id \in \mathsf{H}$, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{c}}(id, v_0, v_1)$ and obtains some b. $\mathcal{B}$ records $(id, (id, b))$ in a list $\mathsf{BB}_{\mathsf{init}}$, and stores (id, v_0, v_1, b) in a list V. $\mathcal{B}$ then returns (id, b) to $\mathcal{A}$. At some point, $\mathcal{A}$ returns to $\mathcal{B}$ some board BB.

Note that

- By construction, during the execution, the list $[b|(id, (p, b)) \in \mathsf{BB}_{\mathsf{init}}]$ of the ciphertexts in $\mathsf{BB}_{\mathsf{init}}$ (kept by $\mathcal{B}$) is always equal to the list L in the game $\mathsf{Exp}_{\mathcal{V}}^{\mathsf{NM},\beta'}$ played by $\mathcal{B}$.
- It is also clear by examining the game $\mathsf{Exp}_{\mathcal{A}}^{\text{mb-BPRIV}1,\beta}$ that, up to this point, $\mathcal{A}$ has been accurately simulated, and has the same view it would have in game $\mathsf{Exp}_{\mathcal{A}}^{\text{mb-BPRIV}1,\beta}$.
- By construction of the $\mathcal{O}\mathsf{voteLR}$ oracle, the list $\mathsf{BB}_{\mathsf{init}}$ is also equal to the list BB_β in (the corresponding execution of) the game $\mathsf{Exp}_{\mathcal{A}}^{\text{mb-BPRIV}1,\beta}$ for $\mathcal{A}$.
- We can also note that $\mathsf{BB}_{\mathsf{init}}$ contains no duplicate ciphertexts (except with a negligible probability $p_1^\beta(\lambda)$). Indeed, $\mathsf{BB}_{\mathsf{init}}$ containing duplicate ciphertexts would imply that two oracle calls $\mathsf{Vote}(\mathsf{pk}, id, v)$ and $\mathsf{Vote}(\mathsf{pk}, id', v')$ produced ballots with the same ciphertext. If this happened with non-negligible probability, by submitting $\mathcal{O}_{\mathsf{c}}(id, v, v'')$ followed by $\mathcal{O}_{\mathsf{c}}(id', v', v''')$ (for some $v'' \neq v'''$) to the encryption oracle, and checking whether the ciphertexts of the two resulting ballots are equal, an adversary would win the non-malleability game $\mathsf{Exp}^{\mathsf{NM}'}$. The two ciphertexts obtained on the right would indeed be different, except with negligible probability, by strong consistency (as they contain different votes).
- Finally, by assumption $\mathcal{A}$ is only allowed at most one call to $\mathcal{O}\mathsf{voteLR}$ for each id. Thus $\mathsf{BB}_{\mathsf{init}}$, as well as V, contain at most one entry for each distinct id.

Once $\mathcal{A}$ has returned BB, $\mathcal{B}$ checks (in V) that all voters in $\mathsf{H}_{\mathsf{check}}$ have voted, and halts if not. $\mathcal{B}$ also checks whether $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$. If not, $\mathcal{B}$ directly asks $\mathcal{A}$ to guess the bit β', and returns $\mathcal{A}$'s guess.

Following the structure of game $\mathsf{Exp}^{\text{mb-BPRIV}1}$, $\mathcal{B}$ then lets $\mathcal{A}$ perform the verifications for the honest voters. That is, when $\mathcal{A}$ calls $\mathcal{O}\mathsf{verify}_{\mathsf{BB}}(id)$, $\mathcal{B}$ retrieves the entry (id, v_0, v_1, b) for id in V, and checks that b occurs in BB, *i.e.* that $\exists p.\ (p, b) \in \mathsf{BB}$. Once $\mathcal{A}$ is done with the verification phase, $\mathcal{B}$ checks that each $id \in \mathsf{H}_{\mathsf{check}}$ has verified (and halts otherwise). If any of the verifications have failed, $\mathcal{B}$ directly asks $\mathcal{A}$ to guess the bit β', and returns $\mathcal{A}$'s guess.

If all verifications are successful, $\mathcal{B}$ will simulate the tallying phase to $\mathcal{A}$. First, $\mathcal{B}$ computes $\pi = \textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}{}_{\mathsf{U}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB})$.

Note, at this point, that since all checks succeeded, π is simpler than in the general case. Indeed, this means that the algorithm $\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}{}_{\mathsf{U}}(\mathsf{BB}_{\mathsf{init}}, \mathsf{BB})$ did not need to add back any ballots from a honest voter who checked and whose ciphertext would be missing from BB. That is, the list $L' = [i|\mathsf{BB}_{\mathsf{init}}[i] = (id, (p, b)) \ \wedge \ id \in \mathsf{H}_{\mathsf{check}} \ \wedge \ \forall p'.\ (p', b) \notin \mathsf{BB}]$ that $\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}$ constructs is empty.

Hence, π is only defined on $[\![1, |\mathsf{BB}|]\!]$, and, except with negligible probability $p_2^\beta(\lambda)$, is such that for all i:

- $\pi(i) = j$ if $\mathsf{BB}[i] = (p, b)$ and j is the index of the first (and only) occurrence of b in $\mathsf{BB}_{\mathsf{init}}$,
- or $\pi(i) = (id, b)$, if $\mathsf{BB}[i] = (p, b)$ if no such index exists, where all the ids are distinct identities belonging to either a dishonest voter, or a honest voter in $\mathsf{H}_{\overline{\mathsf{check}}}$ who does not have a ciphertext (in $\mathsf{BB}_{\mathsf{init}}$) that appears in BB.

  Indeed, let us show that the test "$S \neq \emptyset$" performed by $\textsc{Recover}^{\mathsf{del,reorder,change}'}$ when adding a cast ballot to L always succeeds. Keeping the notations from the definition of $\textsc{Recover}^{\mathsf{del,reorder,change}'}$, let m be the number of ballots to be cast in L, that is, the number of ciphertexts from BB that do not occur in $\mathsf{BB}_{\mathsf{init}}$. Let also $\overline{S} = \mathsf{D} \cup \{id \in \mathsf{H}_{\overline{\mathsf{check}}} \mid \forall p, b.\ (id, (p, b)) \in \mathsf{BB}_{\mathsf{init}} \Rightarrow \forall p'.\ (p', b) \notin \mathsf{BB})\}$ be the initial value of S. Let then S' be the set of identities occurring in BB that are also in $\overline{S}$, and S'' the set of identities in BB but not in $\overline{S}$. Since $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$, by definition, all identities in BB are distinct, and all ciphertexts also are.

  For all $id \in S''$, id appears in BB, and either $id \in \mathsf{H}_{\mathsf{check}}$, or $id \in \mathsf{H}_{\overline{\mathsf{check}}}$ has a ciphertext in $\mathsf{BB}_{\mathsf{init}}$ that appears in BB. Actually, since all verifications succeed and since all voters in $\mathsf{H}_{\mathsf{check}}$ have ballots in $\mathsf{BB}_{\mathsf{init}}$, in any case, id's ciphertext appears in BB. Hence each identity in S'' appears in BB, has a ciphertext in $\mathsf{BB}_{\mathsf{init}}$, and this ciphertext appears in BB.

  All of these ciphertexts are distinct, as $\mathsf{BB}_{\mathsf{init}}$ contains no duplicates. There can be at most $|\mathsf{BB}| - m$ such ciphertexts, by definition of m. Hence, $|S''| \leq |\mathsf{BB}| - m$. Therefore, since BB does not contain duplicate identities, $|S'| \geq m$, which implies that $|S| \geq m$.

  Thus, following its definition, $\textsc{Recover}^{\mathsf{del,reorder,change}'}$ keeps in π all the cast ciphertexts, associated with distinct identities from $\overline{S}$.

To continue the simulation of $\mathcal{A}$, $\mathcal{B}$ must then provide $\mathcal{A}$ with the result of the election, to answer $\mathcal{A}$'s calls to $\mathcal{O}\mathsf{tally}'$. $\mathcal{B}$ asks for the decryption of the list of all b such that $\exists j, p.\ \pi(j) = (p, b)$. That is, $\mathcal{B}$ calls $\mathcal{O}_{\mathsf{d}}([b | \exists j, p.\ \pi(j) = (p, b)])$. This call to the decryption oracle is allowed: indeed, by definition of $\textsc{Recover}^{\mathsf{del,reorder,change}'}$, a ciphertext b such that $\pi(j) = (p, b)$ cannot occur in an element of $\mathsf{BB}_{\mathsf{init}}$, and hence is not in L.

$\mathcal{B}$ then constructs a list $\overline{\mathsf{BB}}$ containing the votes in $\overline{\pi}\mathsf{BB}_0$ in clear, BB_0 being the board of the left ballots, maintained by $\mathcal{O}\mathsf{voteLR}$ in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV}^{1,\beta}}_{\mathcal{A},\mathcal{V}}$. This is done by retrieving the vote from the list V for ballots coming from the encryption oracle, and using the decryption oracle for the others. Formally this list is obtained by, for all i:

- $\overline{\mathsf{BB}}[i] = (id, v_0)$ if $\mathsf{BB}[i] = (p, b)$ for some p, b such that $\mathsf{BB}_{\mathsf{init}}[j] = (id, (id, b))$ for some j, and $\mathsf{V}[j] = (id, v_0, v_1, b)$ (again, such a j is then unique, and $\pi(i) = j$).
- $\overline{\mathsf{BB}}[i] = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \pi(i))$, if $\mathsf{BB}[i] = (p, b)$ for some b not occurring in $\mathsf{BB}_{\mathsf{init}}$. In that case $\pi(i) = (id, b)$ for some $id \in \overline{S}$, and $\mathcal{B}$ computes $\overline{\mathsf{BB}}[i]$ as $(id, \mathsf{extract}_{\mathsf{v}}(\mathsf{sk}, b))$, obtaining the extraction of b from the answer of the decryption oracle.

At this point, we have $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))$, except with negligible probability. Indeed, for any i, if we denote $\mathsf{BB}[i] = (id, b)$, we have:

- Either b does not appear in $\mathsf{BB}_{\mathsf{init}}$, and, by definition, $\pi(i) = (id', b)$ for some $id' \in \overline{S}$. Hence $\overline{\pi}(\mathsf{BB}_0)[i] = (id', b)$. Then $\overline{\mathsf{BB}}[i] = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, id', b)$ holds by construction of $\overline{\mathsf{BB}}$.
- Or $\mathsf{BB}_{\mathsf{init}}[j] = (id', (id', b))$ for some j, id' (which are then unique), and, by definition, $\pi(i) = j$. Hence $\overline{\pi}(\mathsf{BB}_0)[i] = (id', b)$. Let (id', v_0, v_1, b) denote $\mathsf{V}[j]$. By construction by

the $\mathcal{O}\mathsf{voteLR}$ oracle, $\mathsf{BB}_0[j] = (id', (id', b))$, where (id', b) is a ballot created by calling $\mathsf{Vote}(\mathsf{pk}, id', v_0)$. By construction of $\overline{\mathsf{BB}}$, $\overline{\mathsf{BB}}[i] = (id', v_0)$. Hence,

$$\begin{array}{rl} & \mathbb{P}(\overline{\mathsf{BB}}[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, id', b))) \\ \leq & \mathbb{P}((\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda);\ \mathsf{U} \leftarrow \mathsf{Register}(1^\lambda, \mathcal{I}); \\ & \quad (s, p, b) \leftarrow \mathsf{Vote}(\mathsf{pk}, id', v_0);\ \mathsf{extract}(\mathsf{sk}, \mathsf{U}, p, b) \neq (id', v_0)) \end{array}$$

which is negligible by strong consistency.

Since $\overline{\mathsf{BB}}$ has as many elements as BB, it is of polynomial size (bounded by the running time of $\mathcal{A}$). Hence, $\mathbb{P}(\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))) = \mathbb{P}(\exists i.\ \overline{\mathsf{BB}}[i] \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0)[i]))$ is also negligible. We write

$$p_3^\beta(\lambda) = \mathbb{P}(\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0)) \text{ in } \mathsf{Exp}_{\mathcal{B},\mathcal{V}}^{\mathsf{NM},\beta}).$$

$\mathcal{B}$ then computes $\rho(\overline{\mathsf{BB}})$, and shows this result to $\mathcal{A}$. $\mathcal{A}$ finally outputs a bit β', that $\mathcal{B}$ returns.

We now argue that $\rho(\overline{\mathsf{BB}})$ is indeed the result $\mathcal{A}$ would have been shown in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},\beta}$.

- If $\beta = 1$, $\mathsf{BB}_{\mathsf{init}} = \mathsf{BB}_1$, and thus $\pi = \textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}{}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})$. As previously explained, we then have

  $$\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}{}_{\mathsf{U}}(\mathsf{BB}_1, \mathsf{BB})}(\mathsf{BB}_0)),$$

  that is, $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')$, where BB' is the board computed using algorithm $\textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}$ in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},1}$. By definition of $\mathcal{O}\mathsf{tally}'$, $\rho(\overline{\mathsf{BB}})$ is thus the election result $\mathcal{A}$ sees in this game.

- If $\beta = 0$, $\mathsf{BB}_{\mathsf{init}} = \mathsf{BB}_0$, and thus $\pi = \textsc{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}{}_{\mathsf{U}}(\mathsf{BB}_0, \mathsf{BB})$.

  By definition of $\mathcal{O}\mathsf{tally}'$, $\mathcal{A}$ receives the result $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}))$ in $\mathsf{Exp}_{\mathcal{A},\mathcal{V}}^{\mathsf{mb\text{-}BPRIV1},1}$.

  Recall that, as explained before, for all i, if $\mathsf{BB}[i] = (id, b)$, then.

  - either b occurs in some element $\mathsf{BB}_0[j]$ (j is then unique), and then $\pi(i) = j$;
  - or $\pi(i) = (id', b)$, where all of the id' are chosen distinct in the set $\overline{S}$ of identities that are either dishonest, or are in $\mathsf{H}_{\overline{\mathsf{check}}}$ and do not have a ciphertext in BB_0 that appears in BB.

  Therefore, the lists of ciphertexts in BB and $\overline{\pi}(\mathsf{BB}_0)$ are equal. Let us then denote $\mathsf{BB} = (id_1, b_1), \ldots, (id_n, b_n)$ and $\overline{\pi}(\mathsf{BB}_0) = (id'_1, b_1), \ldots, (id'_n, b_n)$. As previously explained, we have $\overline{\mathsf{BB}} = \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}(\mathsf{BB}_0))$. Thus

  $$\rho(\overline{\mathsf{BB}}) = \rho((id'_1, \mathsf{extract}_{\mathsf{v}}(\mathsf{sk}, b_1)), \ldots, (id'_n, \mathsf{extract}_{\mathsf{v}}(\mathsf{sk}, b_n))).$$

  Since $\mathsf{ValidBoard}(\mathsf{BB}, \mathsf{pk}) = \top$, all the id'_i are pairwise distinct. In addition, by construction of $\overline{\pi}$, the id_i are comprised of, on one hand, pairwise distinct identities picked in $\overline{S}$, and on the other hand, identities from BB_0, whose associated ciphertext (in BB_0) is present in BB. By construction of $\overline{S}$, these two sets are disjoint, and since, as noted before, there is no revote, the ids in BB_0 are also pairwise distinct. Thus, all the id_i are pairwise distinct.

Hence, there exists a permutation of identities that maps each id_i on id'_i. By assumption on ρ, we therefore have

$$\rho((id_1, \mathsf{extract_v}(\mathsf{sk}, b_1)), \ldots, (id_n, \mathsf{extract_v}(\mathsf{sk}, b_n))) = \rho((id_1, \mathsf{extract_v}(\mathsf{sk}, b_1)), \ldots, (id_n, \mathsf{extract_v}(\mathsf{sk}, b_n))),$$

i.e.

$$\rho(\overline{\mathsf{BB}}) = \rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})).$$

By definition of $\mathcal{O}\mathsf{tally}'$, $\rho(\overline{\mathsf{BB}})$ is thus the election result $\mathcal{A}$ sees in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}$.

As we argued, except if

- $\mathsf{BB}_{\mathsf{init}}$ contains duplicate ciphertexts ($p_1^\beta(\lambda)$),
- or π does not have the expected form ($p_2^\beta(\lambda)$),
- or $\overline{\mathsf{BB}} \neq \mathsf{extract}(\mathsf{sk}, \mathsf{U}, \overline{\pi}\mathsf{BB}_0)$ ($p_3^\beta(\lambda)$),

$\mathcal{A}$ is run until the end of its execution by $\mathcal{B}$ if and only if it would also reach the end of the game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A}}$; and $\mathcal{A}$ run by $\mathcal{B}$ has the same view it would have in the corresponding execution of $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$. Hence, except in those three cases, $\mathcal{B}$ returns 1 in $\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{B}}$ if and only if $\mathcal{A}$ returns 1 in the corresponding execution of $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A}}$. Thus, for any β, if $p^\beta(\lambda)$ denotes $p_1^\beta(\lambda) + p_2^\beta(\lambda) + p_3^\beta(\lambda)$,

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},\beta}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \leq p^\beta(\lambda).$$

Therefore

$$\begin{aligned}
& |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
\leq\ & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0'}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1'}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0'}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1'}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
\leq\ & |\mathbb{P}(\mathsf{Exp}^{\mathsf{NM},0'}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{NM},1'}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| + p^0(\lambda) + p^1(\lambda),
\end{aligned}$$

which implies that, assuming that strong consistency holds, and that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{NM}'}$, no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$.

The remaining steps of the proof are identical to the case of Belenios and Civitas.

$\underline{\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}} \wedge \mathsf{Exp}^{\mathsf{SC}} \Rightarrow \mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}.}$

We now show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$. Consider the same $\mathcal{A}$ playing $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1}}$ instead. $\mathcal{A}$ has the same view in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},\beta}_{\mathcal{A},\mathcal{V}}$ as in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},\beta}_{\mathcal{A},\mathcal{V}}$, except if the result r returned by the Tally algorithm differs from ρ applied to the extractions of the board, which happens only with negligible probability by the second point of the strong consistency assumption.

More precisely, consider an adversary $\mathcal{B}$ playing $\mathsf{Exp}^{\mathsf{SC}}$, that runs $\mathcal{A}$ internally, simulating its execution in game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}$. $\mathcal{B}$ answers to any call to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ made by $\mathcal{A}$ by running $\mathsf{Vote}(\mathsf{pk}, id, v_0)$, and returning the generated ballot to $\mathcal{A}$. When $\mathcal{A}$ produces a board BB, $\mathcal{B}$ returns this board.

When $\beta = 0$, $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}$ give $\mathcal{A}$ the same view, and hence return the same result, except if $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB})) \neq r$, where $(r, \Pi) = \mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$. That is, except if $\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{B},\mathcal{V}}$ returns 1. Therefore

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \leq \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{B},\mathcal{V}}(\lambda) = 1).$$

Similarly, consider an adversary $\mathcal{C}$ playing $\mathsf{Exp}^{\mathsf{SC}}$, that runs $\mathcal{A}$ internally, simulating its execution in game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}$. $\mathcal{C}$ answers to any call to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ made by $\mathcal{A}$ by running $\mathsf{Vote}(\mathsf{pk}, id, v_0)$ and $\mathsf{Vote}(\mathsf{pk}, id, v_1)$, storing the generated ballots in boards BB_0 and BB_1, and returning the ballot for v_1 to $\mathcal{A}$. When $\mathcal{A}$ produces a board BB, $\mathcal{C}$ computes $\mathsf{BB}' = \overline{\textsc{Recover}_{\mathsf{U}}(\mathsf{BB}, \mathsf{BB}_1)}(\mathsf{BB}_0)$, and returns this board.

When $\beta = 1$, $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}$ give $\mathcal{A}$ the same view, and hence return the same result, except if $\rho(\mathsf{extract}(\mathsf{sk}, \mathsf{U}, \mathsf{BB}')) \neq r$, where $(r, \Pi) = \mathsf{Tally}(\mathsf{BB}', \mathsf{sk})$. That is, except if $\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}$ returns 1. Therefore

$$|\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \leq \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}(\lambda) = 1).$$

Therefore:

$$\begin{array}{rl} & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & + |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & + |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ \leq & |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV1},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\ & + \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) + \mathbb{P}(\mathsf{Exp}^{\mathsf{SC}}_{\mathcal{C},\mathcal{V}}(\lambda) = 1), \end{array}$$

which implies, by the strong consistency assumption, that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$.

$\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}} \wedge$ different random oracles $\Rightarrow \mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$.

We now show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$. Consider an adversary $\mathcal{B}$ against $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2}}$, that runs $\mathcal{A}$ internally. During the voting phase, $\mathcal{B}$ answers $\mathcal{A}$'s oracle queries by making the same calls to its oracles, and once $\mathcal{A}$ returns a board BB, $\mathcal{B}$ returns this same board. $\mathcal{B}$ also runs its own random oracle, that is made available to $\mathcal{A}$, to simulate the random oracle used for the proof of correct tallying that $\mathcal{A}$ expects to receive. This is made possible by the assumption that this random oracle is not used in any other part of the protocol. Once $\mathcal{A}$ calls $\mathcal{O}\mathsf{tally}'''$, $\mathcal{B}$ calls $\mathcal{O}\mathsf{tally}''$ and receives a result r. Manipulating the random oracle it runs, $\mathcal{B}$ then produces a proof $\Pi = \mathsf{SimProof}(r, \mathsf{BB})$, and returns (r, Π) to $\mathcal{A}$. When $\mathcal{A}$ makes its guess regarding β, $\mathcal{B}$ returns the same guess.

It is clear that $\mathcal{A}$ as run by $\mathcal{B}$ has the same view it would have in game $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},\beta}_{\mathcal{A},\mathcal{V}}$. Hence

$$\begin{array}{c} |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| = \\ |\mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV2},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \end{array}$$

is negligible, which proves no adversary has a non-negligible advantage in $\mathsf{Exp}^{\mathsf{mb\text{-}BPRIV3}}$.

$\underline{\mathsf{Exp}^{\textsf{mb-BPRIV3}} \wedge \mathsf{Exp}^{\mathsf{ZK}} \Rightarrow \mathsf{Exp}^{\textsf{mb-BPRIV}}.}$

Finally we show that no adversary has a non-negligible advantage in $\mathsf{Exp}^{\textsf{mb-BPRIV}}$. Let $\mathcal{A}$ be an adversary against $\mathsf{Exp}^{\textsf{mb-BPRIV}}$. Consider $\mathcal{A}$ playing $\mathsf{Exp}^{\textsf{mb-BPRIV3}}$ instead.

It is clear that, when $\beta = 1$, $\mathsf{Exp}^{\textsf{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\textsf{mb-BPRIV},1}_{\mathcal{A},\mathcal{V}}$ are identical. Hence

$$\mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV},1}_{\mathcal{A},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}} = 1).$$

When $\beta = 0$, the outputs of $\mathsf{Exp}^{\textsf{mb-BPRIV},1}_{\mathcal{A},\mathcal{V}}$ and $\mathsf{Exp}^{\textsf{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}$ can only differ if $\mathcal{A}$ is able to distinguish the real proof of correct tallying Π returned by $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$ from the simulated proof $\mathsf{SimProof}(r, \mathsf{BB})$.

More precisely, let $\mathcal{B}$ be an adversary against $\mathsf{Exp}^{\mathsf{ZK},\beta'}_{\mathcal{V}}$. $\mathcal{B}$ runs $\mathsf{Exp}^{\textsf{mb-BPRIV},0}_{\mathcal{A},\mathcal{V}}$ internally. That is, $\mathcal{B}$ generates its own sets of credentials U, using the $\mathsf{Register}$ algorithm, and runs $\mathcal{A}$, answering calls to $\mathcal{O}\mathsf{voteLR}(id, v_0, v_1)$ by computing $\mathsf{Vote}(\mathsf{pk}, id, v_0)$. $\mathcal{B}$ obtains a board BB from $\mathcal{A}$, on which it performs the validity checks, and the voter verifications, using $\mathsf{ValidBoard}$. If this check fails, $\mathcal{B}$ answers 0. Otherwise, it lets $\mathcal{A}$ call its verification oracle, using Verify to answer $\mathcal{A}$'s queries. Again, if $\mathcal{A}$ does not make all voters in $\mathsf{H}_{\mathsf{check}}$ verify, $\mathcal{B}$ answers 0. If some verifications fail, $\mathcal{B}$ asks $\mathcal{A}$ for its guess regarding the bit β', and returns it. Otherwise, $\mathcal{B}$ returns the board BB, and obtains (r, Π_β), where r is the result of the tally, and $\Pi_{\beta'}$, depending on β', is either the real proof Π_0 computed by $\mathsf{Tally}(\mathsf{BB}, \mathsf{sk})$, or the simulated proof $\Pi_1 = \mathsf{SimProof}(r, \mathsf{BB})$. $\mathcal{B}$ continues to run $\mathcal{A}$, answering (r, Π_β) when $\mathcal{A}$ calls $\mathcal{O}\mathsf{tally}$. Finally $\mathcal{A}$ makes a guess regarding β', that $\mathcal{B}$ returns as its output.

It is clear that, if $\beta' = 0$, $\mathcal{A}$ as simulated by $\mathcal{B}$ in $\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}$ has the same view it would have in $\mathsf{Exp}^{\textsf{mb-BPRIV},0}_{\mathcal{A},\mathcal{V}}$. Hence, $\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV},0}_{\mathcal{A},\mathcal{V}} = 1)$.

Similarly if $\beta' = 1$, $\mathcal{A}$ simulated by $\mathcal{B}$ in $\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}$ has the same view it would have in $\mathsf{Exp}^{\textsf{mb-BPRIV3},0}_{\mathcal{A},\mathcal{V}}$. Hence, $\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}} = 1) = \mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV3},0}_{\mathcal{A},\mathcal{V}} = 1)$.

Thus,

$$\begin{aligned}
& |\mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
= \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
\leq \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)| \\
= \; & |\mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},0}_{\mathcal{B},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\mathsf{ZK},1}_{\mathcal{B},\mathcal{V}}(\lambda) = 1)| \\
& + |\mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV3},0}_{\mathcal{A},\mathcal{V}}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{\textsf{mb-BPRIV3},1}_{\mathcal{A},\mathcal{V}}(\lambda) = 1)|,
\end{aligned}$$

which implies that, assuming that the zero-knowledge property holds, no adversary has a non-negligible advantage in $\mathsf{Exp}^{\textsf{mb-BPRIV}}$. □

B.5.5 Ideal functionality corresponding to $\mathsf{Recover}^{\mathsf{del},\mathsf{reorder},\mathsf{change}'}$

Recall the predicate $P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}$ from the functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho)$:

$\underline{P^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(L, f) = \top \text{ iff:}}$

- f keeps the votes of all voters who check:
  $\forall i.\, \forall id \in \mathsf{H}_{\mathsf{check}}.\, \forall v.\, L[i] = (id, v) \implies \exists j.\, f(j) = i.$
- and no votes from voters who check are modified by f:
  $\forall i, id, v.\, f(i) = (id, v) \implies id \in \mathsf{D} \cup \mathsf{H}_{\overline{\mathsf{check}}}.$

Theorem B.23. *The* $\textsc{Recover}^{\mathsf{del,reorder,change}'}$ *algorithm defined above is compatible with the predicate* $P^{\mathsf{del,reorder,change}}$.

Proof. Indeed, consider $\mathcal{A}$ that plays the game $\mathsf{Exp}^{\mathsf{comp},P^{\mathsf{del,reorder,change}},\textsc{Recover}^{\mathsf{del,reorder,change}'}}_{\mathcal{A},\mathcal{V}}$. Following this game, let $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, $\mathsf{U} \leftarrow \mathsf{Register}(1^\lambda,\mathcal{I})$, $\mathsf{CU} \leftarrow [\mathsf{U}[id]|id \in \mathsf{D}]$, and $\mathsf{PU} \leftarrow [\mathsf{Pub}(\mathsf{U}[id])|id \in \mathcal{I}]$. $\mathcal{A}$ has access to the $\mathcal{O}\mathsf{vote}$ oracle, to generate honest ballots, that get stored in a board BB_1. For each $(id,(p,b)) \in \mathsf{BB}_1$, by construction, (p,b) was constructed by calling $\mathsf{Vote}(\mathsf{pk},id,\mathsf{U}[id],v)$ for some v. Let BB be the board returned by $\mathcal{A}$ in the game. Note that, by the non-malleability assumption on ciphertexts, following the same reasoning as in the previous proof, BB_1 contains no duplicate ciphertexts, except with negligible probability. If BB is not valid, or BB_1 does not contain at least one entry for each $id \in \mathsf{H}_{\mathsf{check}}$, $\mathcal{A}$ loses the game.

Otherwise let $\pi = \textsc{Recover}^{\mathsf{del,reorder,change}}_{\mathsf{U}}(\mathsf{BB}_1,\mathsf{BB})$. Let us show that π is compatible with $P^{\mathsf{del,reorder,change}}$ *w.r.t.* $\mathsf{sk},\mathsf{U},L_{id}$, except with negligible probability. Assume BB_1 contains no duplicate ballots, which, as explained, holds with overwhelming probability.

Let L be a list of elements (id,v), such that $[id|(id,v) \in L] = L_{id}$. We show that $P^{\mathsf{del,reorder,change}}(L,\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)) = \top$.

Consider the lists LL, LL', LL'' constructed by the following process:

$LL \leftarrow []$
$S \leftarrow \mathsf{D} \cup \{id \in \mathsf{H}_{\overline{\mathsf{check}}} \mid \forall p,b.\ (id,(p,b)) \in \mathsf{BB}_1 \ \Rightarrow\ \forall p'.\ (p',b) \notin \mathsf{BB})\}$
for $(p,b) \in \mathsf{BB}$ **do**
  if $\exists j, id.\ \mathsf{BB}_1[j] = (id,(id,b))$ **then**
    $LL \leftarrow LL \parallel j$
    (by assumption on BB_1, this j is unique)
  else if $S \neq \emptyset$ **then**
    $LL \leftarrow LL \parallel \mathsf{extract}(\mathsf{sk},\mathsf{U},id,b); S \leftarrow S \backslash \{id\}$
    (for some $id \in S$)
$LL' \leftarrow [i|\mathsf{BB}_1[i] = (id,(p,b)) \ \wedge\ id \in \mathsf{H}_{\mathsf{check}} \ \wedge\ \forall p'.\ (p',b) \notin \mathsf{BB}]$
$LL'' \leftarrow LL \parallel LL'$
return $(\lambda i.\ LL''[i])$

Let us also denote $\overline{S} = \mathsf{D} \cup \{id \in \mathsf{H}_{\overline{\mathsf{check}}} \mid \forall p,b.\ (id,(p,b)) \in \mathsf{BB}_1 \ \Rightarrow\ \forall p'.\ (p',b) \notin \mathsf{BB})\}$ the initial value of S.

By definition, $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$ is $\lambda i.LL'''[i]$, where LL''' is the list obtained obtained by removing all $\perp$ elements from LL''.

We have to show that $P^{\mathsf{del,reorder,change}}(L,\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)) = \top$. That is, that

1. No votes from voters in $\mathsf{H}_{\mathsf{check}}$ are modified by $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$:

$$\forall i.\ \forall (id,v).\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)[i] = (id,v) \implies id \in \mathsf{D} \cup \mathsf{H}_{\overline{\mathsf{check}}}.$$

   Let i, id, v be such that $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)[i] = (id,v)$. Hence $LL[j] = (id,v)$ for some j. Thus, by construction of LL, $id \in \overline{S}$. Since $\overline{S} \subseteq \mathsf{D} \cup \mathsf{H}_{\overline{\mathsf{check}}}$ by definition, we have $id \in \mathsf{D} \cup \mathsf{H}_{\overline{\mathsf{check}}}$.

2. $\mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)$ keeps the votes of all voters who check:

$$\forall i.\ \forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall v.\ L[i] = (id,v) \implies \exists j.\ \mathsf{mod}_{\mathsf{sk},\mathsf{U}}(\pi)(j) = i,$$

   that is,

$$\forall id \in \mathsf{H}_{\mathsf{check}}.\ \forall i,b.\ \mathsf{BB}_1[i] = (id,(id,b)) \implies \exists j.\ LL''(j) = i.$$

   Let $id \in \mathsf{H}_{\mathsf{check}}, i, b$ such that $\mathsf{BB}_1[i] = (id,(id,b))$. We distinguish two cases:

- either $\forall p'.\ (p', b) \notin \mathsf{BB}$: in that case, by definition of LL', $i \in LL'$. Hence, since LL' is a sublist of LL'', $i \in LL''$ and the claim holds.
- or there exists j such that $\mathsf{BB}[j] = (p', b)$ for some p'. In that case, by construction of LL, we have $LL[j] = i$, and thus $LL''[j] = i$.

In any case, the claim holds, which concludes the proof.

□

B.5.6 Conclusion on Helios without revote

Theorem B.24. *Assume Helios, with no revote allowed, is strongly consistent, has non-malleable ciphertexts, that the proofs of correct tallying are zero-knowledge, that different random oracles are used for these proofs and the rest of the protocol, and that the counting function ρ is id-blind.*

Then Helios securely implements the ideal functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho)$, when considering only environments that do not make voters revote.

Proof. It directly follows from Theorems 12, B.22, and B.23 that, under these assumptions, Helios securely implements the functionality $\mathcal{F}_{\mathsf{v}}^{\mathsf{del},\mathsf{reorder},\mathsf{change}}(\rho)$. □

Résumé en français

1 Contexte

Le vote électronique a pris une importance croissante ces dernières années. Les technologies de l'information ont une présence grandissante dans nos sociétés, et une application naturelle est d'envisager leur utilisation pour faciliter la tenue d'élections. Le vote électronique prend deux formes principales. La première est l'utilisation de machines à voter, placées dans les bureaux de votes. La seconde, à laquelle nous nous intéressons dans cette thèse, est celle des systèmes de vote à distance, où les électeurs votent par Internet.

Les avantages du vote électronique sont manifestes : il rend l'organisation d'élections bien plus simple et pratique. En particulier, il est plus facile de collecter et comptabiliser les votes : les bulletins sont envoyés par Internet à un serveur contrôlé par les autorités, puis comptés de façon informatique. Cela pourrait aider à réduire les infrastructures nécessaires pour organiser des élections ou référendums. Des systèmes de vote électronique, sur place ou à distance, ont déjà été utilisés dans de vraies élections tant à petite qu'à grande échelle, par exemple le vote par Internet a été utilisé pour des élections législatives en Estonie ou des référendums en Suisse, et des machines à voter ont été utilisées pour des élections nationales aux États-Unis ou au Brésil.

Cependant, comme pour tout système manipulant des données sensibles, se pose le problème de s'assurer que ces données sont traitées de façon sécurisée. Comment être certain que le résultat annoncé par les organisateurs à la fin de l'élection est le bon ? Que mon vote a bien été compté ? Puis-je être sûr que les votes restent secrets, et que nul ne peut apprendre pour quel candidat j'ai voté ? Ces questions ont été débattues depuis des siècles dans le cadre du vote "papier" – les systèmes de vote traditionnels. Elles sont généralement considérées comme cruciales du point de vue de la démocratie.

Dans les systèmes de vote papier, il est possible dans une mesure raisonnable d'obtenir de telles garanties. Au bureau de vote, je peux choisir secrètement mon bulletin dans l'isoloir, et le placer dans une enveloppe puis dans l'urne. Je peux ensuite surveiller l'urne pour m'assurer qu'elle n'est pas ouverte avant le dépouillement, ce qui m'assure que mon vote reste secret. Je peux également assister au dépouillement dans mon bureau de vote pour être certain que tous les bulletins, y compris le mien, sont correctement comptés.

Dans le cas de systèmes de vote électronique à distance en revanche, il est souvent bien moins facile d'obtenir ces garanties de sécurité. Cependant la sécurité du système est encore plus cruciale dans ce cas : l'élection étant organisée par Internet, une attaque sur le système pourrait être conduite de n'importe où à grande échelle, alors que pour le vote papier toute manipulation des votes doit être menée dans de nombreux bureaux de votes pour avoir une réelle influence, et est donc plus aisée à détecter.

Protocoles Pour apporter de telles garanties, les systèmes de vote à distance prennent la forme de protocoles cryptographiques. Ils utilisent la cryptographie (chiffrement, signatures, ...) pour protéger le scrutin, et décrivent précisément quelles opérations cryptographiques doivent être effectuées par chaque partie (les votants, les autorités, ...). Hors un problème maintenant bien connu est qu'il est difficile de construire des protocoles cryptographiques correctement – de les rendre réellement sûrs. Un exemple majeur est le protocole TLS (Transport Layer Security), qui a pour but de sécuriser les communications sur le Web. Il apporte notamment des garanties d'authentification, pour que les utilisateurs puissent être sûrs qu'ils communiquent avec le serveur qu'ils voulaient joindre. TLS vise également à garantir la confidentialité et l'intégrité des communications : toute donnée échangée entre un serveur et un client doit rester secrète aux yeux de toute autre tierce personne, et ne doit pas non plus pouvoir être modifiée par

elle.

TLS a été proposé en 1999 [73], puis corrigé et mis à jours plusieurs fois depuis, la dernière version datant de 2018 [92] : avec le temps, de nombreuses failles ont été découvertes dans chaque version successive (*e.g.* Logjam [12], POODLE [84], Triple Handshake [30], ...), ce qui signifie que le protocole n'a jamais réellement été sûr.

Analyse formelle Les méthodes formelles se sont avérées être un outil puissant pour étudier la sécurité des protocoles cryptographiques. De manière générale, elles consistent à construire un modèle mathématique des protocoles, et des adversaires essayant de les attaquer, ainsi que des définitions formelles des propriétés exprimant leur sécurité. L'objectif est alors de prouver qu'aucun adversaire dans le modèle choisi ne peut violer ces propriétés. Durant les dernières décennies, elles ont été utilisées pour prouver sûrs de nombreux protocoles déployés et largement utilisés – ou pour trouver leurs failles. On peut notamment citer le protocole BAC (Basic Access Control) utilisé par les puces RFID des passeports biométriques [14], le protocole AKA (Authenticated Key Agreement) utilisés par les téléphones portables pour se connecter aux réseaux 3G/4G/5G [16, 23, 85], ou TLS mentionné précédemment [72].

Observons à présent un exemple de protocole de vote, pour comprendre à quel type d'attaques ils sont sujets, ainsi que comment les méthodes formelles sont utiles dans ce contexte.

2 Exemple : Helios

Description Helios est un protocole de vote électronique proposé pour la première fois en 2009 [10]. Bien qu'il n'ait pas été déployé à grande échelle pour des élections avec de grands enjeux, il a été utilisé dans de nombreuses situations à plus petite échelle, telles que l'élection du recteur de l'Université de Louvain-la-Neuve [11].

Entre autres, Helios a pour but de protéger la vie privée des votants en gardant leurs votes secrets. Pour cela, il fait usage de cryptographie asymétrique. Les détails techniques de la construction cryptographique utilisée ne sont pas pertinents dans cette description informelle du système. Plusieurs administrateurs – les autorités de l'élection – génèrent une pair de clefs publique et privée. La clef publique est publiée, et la clef privée séparée en plusieurs parts qui sont distribuées entre les administrateurs. Ces parts sont construites de telle sorte que tous les administrateurs (ou du moins une part suffisamment grande d'entre eux) doivent donner leur accord pour déchiffrer un message. Ceci réduit le risque d'utilisation abusive de la clef : il suffit ainsi de faire confiance à une fraction des administrateurs. Les votants utilisent ensuite la clef publique pour chiffrer leurs votes, et les administrateurs doivent conjointement déchiffrer les bulletins pour calculer le résultat de l'élection. Pour plus de clarté, nous feront dorénavant référence au groupe d'administrateur comme s'il s'agissait d'une seule autorité, que nous appellerons autorité de comptage, et qui est le seul dépositaire de la clef de déchiffrement.

La structure d'une élection dans Helios est décrite informellement ci-dessous.

- L'autorité génère des clefs publique et privée $(\mathsf{pk}, \mathsf{sk})$, et publie la clef publique pk. Il s'agit d'une abstraction du protocole réellement utilisé pour générer et distribuer les parts de la clef privée entre les administrateurs.
- Chaque votant construit un bulletin : son vote chiffré par pk.
- Un serveur qui joue le rôle de l'urne collecte les bulletins de chaque votant. Concrètement, les votants se connectent au serveur avec un mot de passe établi au préalable, et envoient leur bulletin.
- La liste des bulletins reçus au fil de l'élection par le serveur est rendue publique au fur et à mesure. En particulier, chaque votant peut s'il le souhaite vérifier que leur bulletin y apparaît bien.
- Enfin, l'autorité de comptage est en charge de calculer le résultat de l'élection : elle récupère tous les bulletins sur la liste publique, les déchiffre (en utilisant la clef secrète pk), puis calcule et publie le résultat.

Attaque Il semble au premier abord que le secret des votes soit respecté, pourvu que l'on fasse confiance à l'autorité de comptage. En effet, les votes envoyés sur le réseau sont toujours chiffrés, et seule

l'autorité possèdent (les parts de) la clef nécessaire au déchiffrement. Il semble donc clair qu'à condition que le chiffrement utilisé soit sûr, un attaquant ne peut jamais avoir accès aux votes en clair.

Cependant, il existe en réalité une attaque contre Helios, décrite initialement dans [65]. Considérons un scénario très simple, celui d'une élection avec seulement trois votants : Alice, qui souhaite voter pour le candidat α, Bob, qui veut voter pour un autre candidat β, et Charlie, qui est en réalité un attaquant, et tente d'apprendre quel est le candidat pour lequel Alice vote. Suivant le protocole, Alice et Bob soumettent à l'urne leurs bulletins respectifs b_A et b_B, contenant leurs votes α, β chiffrés.

Si l'élection prenait fin à ce stade, l'urne contiendrait uniquement b_A et b_B. Le résultat serait donc "une voix pour α, une pour β". En voyant ce résultat, Charlie ne saurait pas si c'est Alice ou Bob qui a voté α : pour apprendre le vote d'Alice, il lui faut effectuer d'autres actions.

Charlie pourrait tout d'abord "espionner" Alice lorsqu'elle envoie son bulletin b_A, de façon à apprendre ce bulletin. Il ne peut bien entendu pas le déchiffrer, n'ayant pas accès à la clef secrète sk. Il peut en revanche soumettre à l'urne une seconde copie de b_A en son propre nom, c'est-à-dire en prétendant qu'il s'agit de son bulletin. La liste des bulletins dans l'urne est alors $[b_A, b_B, b_A]$. L'autorité de comptage va alors calculer le résultat "deux voix pour α, une pour β". Cette fois ci, même si Charlie ne connaît pas le contenu du bulletin qu'il a soumis, il sait qu'il s'agit du même vote qu'Alice. En voyant ce résultat, il peut donc déduire que ce vote ne peut être qu'α, puisque seul ce candidat a obtenu deux voix. Il peut donc apprendre qu'Alice a voté pour α : le secret de son vote est rompu.

Cette attaque peut sembler simpliste et difficile a mener dans une vraie élection avec de nombreux votants. Cependant, les auteurs de [65] ont étudié cette question, et conclu que cette attaque, ou des variantes, aurait en fait été réalisable dans de vraies élections passées, telles que les élections législatives en France, si cette version d'Helios avait été utilisée pour les organiser.

Correction Fort heureusement, cette attaque par copie de bulletin peut aisément être empêchée. Elle repose sur la possibilité de Charlie de soumettre une copie exacte du bulletin d'Alice à l'urne. L'urne pourrait facilement détecter que ce bulletin a déjà été soumis par Alice auparavant, et le rejeter. Cette correction a été proposée par les auteurs de [65]: ils appellent *weeding* l'opération de rejeter les bulletins doublons. Elle permet au moins d'empêcher l'attaque : lorsque Charlie tente de soumettre b_A en son nom, son attaque est détectée, et son bulletin rejeté. Mais suffit elle à rendre Helios entièrement sûr ? Comment s'assurer qu'il n'existe aucune autre manière d'attaquer le secret du vote ? Cette exemple illustre le fait que même des failles très simples sur les protocoles de vote peuvent être difficiles à débusquer. C'est sur ce point que les méthodes formelles peuvent nous aider : en définissant précisément quelles propriétés doivent être satisfaites, il devient possible de vérifier formellement qu'un protocole est sûr.

En réalité, dans le cas d'Helios, il apparaît que le *weeding* échoue à empêcher une autre attaque similaire, découverte par Roenne, dans un contexte où le revote est autorisé, c'est-à-dire où chaque votant peut soumettre plusieurs bulletins, dont seul le dernier est compté.

Cette seconde attaque se déroule comme suit : Charlie décide, non seulement d'espionner, mais aussi de bloquer complètement le bulletin d'Alice – en la déconnectant d'une façon ou d'une autre du serveur. Alice essaiera alors de voter de nouveau, avec un nouveau bulletin b'_A pour le même candidat. Comme auparavant, Charlie soumet b_A en son nom, et malgré le *weeding*, le serveur accepte ce bulletin. En effet, puisque b_A a été bloqué lorsqu'Alice a voulu le soumettre, le serveur n'a jamais reçu ce bulletin, et ne peut donc pas détecter la copie de Charlie. Comme dans l'attaque précédente, le résultat contiendra deux voix pour le choix d'Alice, permettant à Charlie d'apprendre son vote.

3 Propriétés

Plusieurs propriétés caractérisent la sécurité qu'un protocole de vote électronique doit apporter. Nous concentrerons notre étude sur les propriétés suivantes, que l'on peut considérer comme les plus importantes.

Vérifiabilité La vérifiabilité exprime intuitivement l'idée selon laquelle les votants doivent pouvoir se convaincre que le résultat publié est le bon, et n'a pas été altéré. Elle est généralement formalisée comme la conjonction des trois aspects suivants.

- Vérifiabilité individuelle : je peux vérifier que *mon* vote a correctement été placé dans l'urne.

- Vérifiabilité universelle : n'importe qui peut vérifier que le résultat publié par l'autorité correspond au comptage du contenu de l'urne.
- Vérifiabilité de l'éligibilité : n'importe qui peut s'assurer que tous les bulletins de l'urne ont été émis par des votants légitimement autorisés à voter.

Ensemble, ces trois propriétés assurent qu'il est possible de vérifier que le résultat compte correctement tous les votes des votants légitimes, et seulement ceux-ci.

Secret du vote Cette propriété correspond à la garantie pour les votants qu'un attaquant ne peut apprendre leur votes. Cependant, elle ne peut être formalisée en demandant simplement que pour tout votant arbitraire Alice, la *valeur* de son vote soit secrète. En effet, si une telle condition peut être utilisée lorsque de vrais secrets (tels qu'une clef) sont mis en jeu, la valeur v du vote est ici tout simplement l'un des choix publiquement proposés aux votants dans l'élection. Cette valeur elle-même n'est donc pas secrète : le secret réside plutôt dans l'association entre votants et votes. Ce qui doit être secret est le fait que c'est *Alice* (et non un autre votant) qui a voté pour v.

Ce type de secret se formalise habituellement par une propriété d'indistinguabilité entre deux scénarios : l'un où Alice vote en effet pour v, et l'autre où un autre votant, Bob, vote pour v au lieu d'Alice. Plus précisément, on requiert que pour tous votants arbitraires Alice, Bob, et tous votes 0, 1, aucun attaquant ne puisse déterminer en interagissant avec le protocole si Alice vote pour 0 et Bob pour 1, ou l'inverse. Cette propriété d'indistinguabilité signifie que l'attaquant est incapable de savoir qui a voté pour quel candidat.

***Receipt-freeness*/résistance à la coercition** La *receipt-freeness* et la résistance à la coercition sont deux autres propriétés fréquemment requises pour les protocoles de vote. On peut les voir comme des notions plus fortes de secret, puisqu'elles stipulent que mon vote doit rester secret *même si je souhaite moi-même le révéler*. C'est-à-dire que même si je tente de révéler à un tiers quel est mon vote, je dois être incapable de le convaincre que je lui dis la vérité. Plus précisément, la *receipt-freeness* impose qu'après avoir voté, même si un attaquant m'oblige à lui fournir toutes les informations dont je dispose (mon vote, mon bulletin, mes identifiants, tout code de confirmation que j'aurais reçu,...), il ne doit pas pouvoir déterminer si je lui ai indiqué mon vrai vote ou un faux. En d'autres termes, le processus électoral ne doit pas me fournir de reçu pour mon vote, que je pourrais montrer à d'autres. La résistance à la coercition requiert de plus que l'attaquant ne puisse toujours pas apprendre mon vote même s'il dispose de ces mêmes informations *avant* que je ne vote.

Ces propriétés fortes empêchent la coercition et la vente de votes : puisqu'un tiers ne dispose d'aucun moyen de s'assurer que j'ai bien suivi ses instructions, il ne peut m'obliger à voter pour un candidat de son choix (ni par la force, ni en achetant mon vote).

4 Relations entre les propriétés

Une observation intéressante au sujet de ces propriétés est que le secret du vote semble intuitivement s'opposer à la vérifiabilité. En effet, pour satisfaire le secret, le système de vote doit révéler le moins d'informations possible sur les votes choisis par les votants. Bien entendu, le résultat de l'élection donne intrinsèquement une indication sur ces votes, mais le secret du vote requiert que cette fuite inévitable soit la seule information qu'un attaquant puisse apprendre sur les votes individuels. D'un autre côté, pour qu'un protocole soit vérifiable, il doit comprendre des mécanismes permettant aux votants de vérifier que leur vote a été compté. Pour effectuer ces vérifications, les votants ont généralement besoin que certaines informations additionnelles soient publiées par les autorités en plus du résultat, ce qui fait courir le risque de révéler trop d'information sur les votes à un attaquant et de violer le secret des votes.

Pour cette raison, l'intuition semble indiquer que plus un système de vote est conçu avec la vérifiabilité comme objectif, plus il sera difficile pour ce même système de garantir aussi le secret du vote.

En réalité, cette intuition a été formellement prouvée par Chevallier-Mames, Fouque, Pointcheval, Stern et Traoré [49]. Ils ont établit que la vérifiabilité et le secret inconditionnel du vote sont incompatibles. La notion de secret qu'ils considèrent, le secret inconditionnel, est une notion très forte, qui requiert que les votes soient parfaitement secrets même pour un adversaire disposant d'une capacité de calcul non

bornée. En revanche, aucun résultat de la sorte n'est établi dans le cas – habituel pour les protocoles cryptographiques – d'un adversaire polynomial, tel que considéré dans toutes les définitions du secret du vote par jeux [26]. Néanmoins, suivant cette intuition, les régulations et recommandations officielles sur le vote électronique tendent à préférer l'une des deux propriétés à l'autre – par exemple les recommandations de la CNIL en France ne demandaient jusque récemment que le secret du vote [2], bien que de nouveaux textes demandent également la transparence [4] ; alors que la loi suisse demande en premier lieu le secret, puis en second lieu un niveau de vérifiabilité dépendant du nombre de votants qui votent par Internet [3].

5 Analyse formelle

Pour étudier un protocole et déterminer s'il a les propriétés précédemment décrites, un modèle formel du protocole et des propriétés est nécessaire. Il faut également spécifier contre quel modèle d'attaquant les propriétés sont considérées. Plusieurs modèles ont été proposés pour les protocoles cryptographiques. Ils peuvent être classés en deux grandes catégories.

Modèles symboliques L'attaque sur Helios décrite précédemment provient d'une erreur logique dans la spécification du protocole, plutôt que d'une attaque sur les primitives cryptographiques utilisées : elle peut être menée sans casser la cryptographie. Les modèles symboliques se concentrent sur ce type d'attaque : ils considèrent un modèle abstrait et idéalisé de la cryptographie. Alors que la sécurité des primitives réelles repose généralement sur des hypothèses calculatoires (telles que la difficulté de la factorisation ou du logarithme discret), dans les modèles symboliques, les primitives sont supposées parfaites et incassables. Par exemple, une primitive de chiffrement sera supposée ne laisser fuir *aucune* information sur les messages en clair à quiconque ne connaît pas la clef de déchiffrement. Les primitives sont généralement représentées par des symboles de fonctions abstraits, et les messages échangés par les différents agents sont modélisés par des termes abstraits utilisant ces symboles. Les propriétés des primitives cryptographiques, c'est-à-dire les calculs que les agents et l'adversaire peuvent effectuer, sont caractérisés par une théorie équationnelle, qui est un ensemble d'équation décrivant le comportement des primitives.

Par exemple, le chiffrement asymétrique peut être modélisé par un symbole `aenc`, d'arité 2, et un symbole `pk`, d'arité 1 : $\texttt{pk}(k)$ représente la clef publique associée à la clef privée k, et $\texttt{aenc}(M, \texttt{pk}(k))$ représente le chiffrement asymétrique du message M par cette clef publique. Un symbole associé `adec` peut être utilisé pour représenter l'opération de déchiffrement correspondante : $\texttt{adec}(M, k)$ représente l'opération de déchiffrement de M avec la clef privée k.

Le comportement de la primitive sera alors modélisé par l'équation suivante, qui indique que le déchiffrement d'un chiffré par la bonne clef renvoie le message clair :

$$\texttt{adec}(\texttt{aenc}(M, \texttt{pk}(k)), k) = M$$

Cette théorie équationnelle spécifie que le seul calcul qui peut être effectué sur un chiffré $\texttt{aenc}(M, \texttt{pk}(k))$ est son déchiffrement par la clef k. Ceci modélise une primitive de chiffrement parfaite, qui ne peut être cassée, et ne révèle absolument aucune information sur le message clair à quiconque ne possédant pas la clef de déchiffrement.

Les modèles symboliques considèrent généralement un attaquant exerçant un contrôle complet sur le réseau, et qui peut donc intercepter ou bloquer tout message envoyé par un agent du protocole, et, dans les limites imposées par la théorie équationnelle, effectuer tout calcul sur ces messages, en appliquant des symboles de fonctions pour construire de nouveaux termes. Enfin, l'attaquant a la possibilité de modifier les messages en chemin, en les remplaçant par tout terme qu'il peut calculer. Ce modèle d'attaquant, introduit par Dolev and Yao en 1983 [74], est généralement désigné par *attaquant de Dolev-Yao*.

Différents modèles symboliques existent pour représenter l'exécution d'un protocole en présence d'un tel attaquant. L'une des approches les plus répandues, et celle que l'on utilisera dans cette thèse lorsque l'on considèrera un modèle symbolique, est de représenter les protocoles par des processus dans une algèbre de processus telle que le pi-calcul, un outil classique pour la modélisation de programmes distribués. Cette idée a tout d'abord pris la forme du spi-calcul, proposé par Abadi et Gordon en 1997 [8]. Le pi-calcul standard propose des constructions pour modéliser des processus exécutés en parallèle, qui échangent des

messages simples en les envoyant ou recevant sur des canaux de communication. Le spi-calcul l'étend par des primitives cryptographiques, modélisées par des termes comme ci-dessus. Une généralisation, le pi-calcul appliqué, a ensuite été proposée par Abadi et Fournet en [6] : elle sépare davantage la théorie équationnelle de l'algèbre de processus, ce qui rend plus facile la formalisation de propriétés de sécurité de façon indépendante de la théorie équationnelle choisie. Parmi les autres approches, les plus notables sont celles fondées sur la réécriture de multiensembles [42], où les protocoles sont représentés par des règles de réécriture décrivant l'évolution de l'état des participants, et de la connaissance de l'attaquant, au cours de l'exécution.

Modèles calculatoires D'autre part, les modèles calculatoires, explorés dès les années 1980 (par notamment Goldwasser, Micali [79], Blum [36], Yao [97]), considèrent un modèle d'attaquant bien moins abstrait, plus précis, et sans doute plus proche du monde réel. Les protocoles sont représentés par une collection d'algorithmes (*i.e.* de machines de Turing) qui décrivent le comportement de chaque participant, et manipulent des chaînes de bits, plutôt que des termes abstraits. Contrairement au cas symbolique, l'attaquant n'est pas restreint explicitement à certains calculs sur les primitives. Il est plutôt représenté par une machine de Turing probabiliste pouvant effectuer des calculs arbitraires sur les messages qu'elle observe, mais doit s'exécuter en temps polynomial en un *paramètre de sécurité* λ. Ce paramètre peut dénoter la taille des clefs utilisées. Il existe plusieurs façons d'écrire les propriétés dans ces modèles. Les *définitions par simulation* – notamment dans le cadre *Universal Composability* [41] – demandent qu'aux yeux du monde extérieur, même en présence d'un adversaire, le protocole étudié soit indistinguable d'un système idéal dont la sécurité est manifeste. Les *définitions par jeux*, d'autre part, sont exprimés par des *jeux cryptographiques*, dans lesquels un adversaire interagit avec le protocole à travers des oracles qui exécutent les algorithmes qui le modélisent, et doit tenter de casser sa sécurité d'une manière bien précise. Par exemple, une propriété d'indistinguabilité entre deux scénarios se modélise par un jeu $\mathsf{Exp}^{\beta}(\lambda)$, dans lequel l'adversaire interagit avec l'un des deux scénarios possibles, en fonction d'un bit secret β. L'adversaire doit alors deviner la valeur de β, *i.e.* déterminer quel scénario est exécuté : sa proposition constitue le résultat du jeu. L'*avantage* de l'adversaire est alors la différence entre les probabilités qu'il devine 1 lorsque $\beta = 0$ et $\beta = 1$:

$$\mathsf{Adv}_{\mathcal{A}}(\lambda) = |\mathbb{P}(\mathsf{Exp}^{0}(\lambda) = 1) - \mathbb{P}(\mathsf{Exp}^{1}(\lambda) = 1)|$$

La sécurité est définie en demandant qu'asymptotiquement (par rapport au paramètre de sécurité λ) aucun adversaire ne puisse faire significativement mieux que deviner au hasard. C'est-à-dire, qu'aucun adversaire n'ait un avantage $\mathsf{Adv}_{\mathcal{A}}(\lambda)$ non négligeable.

Comme expliqué précédemment, dans le cas du secret du vote, les deux scénarios à distinguer pourraient par exemple être deux exécutions du protocole, dans lesquelles les votes de deux votants sont échangés : dans $\mathsf{Exp}^{0}(\lambda)$ Alice voterait 0 et Bob 1, et dans $\mathsf{Exp}^{1}(\lambda)$ Alice voterait 1 et Bob 0. Une formalisation plus générale laisserait l'adversaire choisir lui même quels votes sont utilisés : il proposerait deux votes v_0, v_1, et, en fonction de β, Alice voterait v_0 et Bob v_1, ou l'inverse. Encore plus généralement, les votes de plus de deux votants peuvent être échangés.

De telles notions de sécurité sont généralement prouvées par réduction, c'est-à-dire en prouvant que briser la sécurité du protocole permettrait également de casser une hypothèse de sécurité sur les primitives cryptographiques sous-jacentes, ce qui à son tour casserait une hypothèse calculatoire (*e.g.* RSA, factorisation, ...).

Les modèles calculatoires sont bien plus précis que les modèles symboliques, car ils ne reposent as sur des abstractions aussi fortes, et font des hypothèses bien plus faibles sur les primitives. En revanche, ces abstractions et idéalisations effectuées par les modèles symboliques se prêtent plus aisément à la vérification automatique, comme discuté dans la section suivante. Dans certains cas cependant, il a été établi que sous des hypothèses raisonnables les modèles symboliques peuvent être calculatoirement corrects, en ce que prouver la sécurité symbolique d'un protocole implique sa sécurité calculatoire [9, 62, 63]. Une autre approche récente qui relie les mondes symbolique et calculatoire est le modèle de Bana-Comon [19]. Il considère une représentation symbolique d'un attaquant sans le restreindre à un ensemble fixé d'opérations, au contraire des modèles symboliques habituels, ce qui lui permet d'être calculatoirement correct par construction, tout en permettant un raisonnement symbolique.

6 Vérification automatique

Symbolique vs calculatoire Prouver manuellement la sécurité d'un protocole, aussi bien dans les modèles symbolique que calculatoire, a tendance à être fastidieux, même pour des exemples simples, et mène facilement à des erreurs qui peuvent être difficiles à détecter. Une idée naturelle est donc de concevoir des procédures et des outils pour produire ces preuves automatiquement. Comme mentionné dans le paragraphe précédent, les modèles calculatoires font usage d'une couche d'abstraction bien plus fine que les modèles symboliques, ce qui rend le raisonnement automatisé plus ardu dans ces modèles. Les seuls outils significatifs pour raisonner dans un cadre calculatoire sont CryptoVerif [32], qui automatise les transformations entre les jeux, EasyCrypt [21], qui fait usage de logique de Hoare probabiliste relationnelle, et F* [96], un langage fonctionnel qui fournit un système de type riche pour encoder des propriétés de sécurité. Seul CryptoVerif est entièrement automatique alors qu'EasyCrypt et F* sont plutôt des outils interactifs qui assistent l'utilisateur dans l'écriture et la vérification d'une preuve.

Nous nous concentrerons plus ici sur les outils de vérification dans le monde symbolique, qui permet une plus grande automatisation.

Propriétés de trace et d'équivalence La vérification symbolique de protocoles cryptographiques s'intéresse à des propriétés qui peuvent généralement être classées en deux grandes catégories. La première est celle des *propriétés de trace*, *i.e.* qui s'expriment comme une condition devant être satisfaite par toute exécution (ou *trace*) du protocole. C'est typiquement le cas de propriétés d'accessibilité, qui requièrent qu'un certain "mauvais" état ne soit jamais atteint. Elles s'utilisent par exemple pour des propriétés faibles de secret : l'exécution n'atteint jamais un état dans lequel l'attaquant est parvenu à apprendre la valeur d'un secret. Dans le cas des protocoles de vote, la vérifiabilité est une propriété de trace : l'exécution ne doit jamais atteindre un point où un résultat ne comptant pas tous les votes est publié.

D'autre part, certaines propriétés dites d'*équivalence* ne peuvent s'exprimer qu'en considérant plusieurs exécutions du protocole. Elles formalisent l'idée que deux situations doivent être indistinguable aux yeux d'un attaquant. Ces propriétés sont très utiles pour exprimer des garanties de vie privée, telles que l'anonymat – un attaquant ne doit pas pouvoir déterminer si c'est moi qui exécute un protocole, ou quelqu'un d'autre. Elles permettent également de formaliser des versions fortes du secret : un attaquant ne doit pas seulement être incapable d'apprendre la valeur d'une donnée secrète (telle qu'une clef), mais doit aussi être incapable de distinguer un protocole utilisant le vrai secret d'une variante utilisant une autre valeur. La non-traçabilité, qui requiert que l'attaquant ne puisse pas déterminer si deux sessions d'un protocoles sont exécutées par le même utilisateur ou non, est un autre exemple de propriété d'équivalence. Le secret du vote, comme expliqué précédemment, est typiquement exprimé comme une propriété d'équivalence.

Des outils automatiques ont tout d'abord été conçus pour vérifier les propriétés de trace. Dans cette thèse nous nous intéressons davantage à la vérification d'équivalences, dont l'étude est plus récente, bien que plusieurs procédures aient été proposées ces dernières années.

Nombre de sessions borné et non borné Les modèles symboliques permettent de considérer un nombre non borné de sessions d'un protocole en parallèle. Vérifier la sécurité dans ce cadre donne des garanties plus fortes : la prouver seulement pour un nombre fixé de session pourrait conduire à rater des attaques qui requièrent un nombre plus grand de sessions pour être menées à bien. Malheureusement, il a été établi que le problème de la déduction – un attaquant peut-il apprendre un message donné en interagissant avec un certain protocole ? – aussi bien que celui de l'équivalence sont indécidables lorsque l'on considère un nombre non borné de sessions [75, 52]. Les outils existants adoptent par conséquent l'une de deux approches possibles : soit restreindre leur cadre d'application à un nombre borné de sessions – auquel cas ils peuvent être des procédures de décision exactes ; soit considérer le cas non borné, et donc être incomplets – typiquement, prouvant les propriétés dans certains cas, mais ne pouvant toujours conclure ou terminer.

Outils Dans le cas non borné, l'un des outils majeurs est ProVerif [31, 33], proposé par Blanchet en 2001 puis amélioré et étendu régulièrement par la suite. ProVerif est entièrement automatique, et prouve des propriétés de (non-)déductibilité en traduisant les protocoles exprimés dans une variante du pi-calcul appliqué en un ensemble de clauses de Horn, qui surapproximent la connaissance qu'un attaquant peut obtenir. Le problème est alors réduit à celui de déterminer si un message donné est déductible de

cet ensemble de clauses. ProVerif a ensuite été étendu [35] aux propriétés d'équivalence de processus ne différant que par les messages échangés, et non leur structure – la *diff-équivalence* – d'une manière similaire. ProVerif est correct, en ce que lorsqu'il déclare un protocole sûr, la propriété est en effet vraie. Il n'est en revanche pas complet : il peut ne pas terminer, ou trouver de fausses attaques qui ne violent pas la propriétés, lorsque sa surapproximation est trop grossière. Par exemple, cette approximation a tendance à traiter tous les processus comme s'ils étaient répliqués, et court donc le risque de trouver de fausses attaques lorsque la sécurité d'un protocole repose sur le fait que certains évènements ne se produisent qu'une fois. C'est notamment le cas dans les protocoles de vote qui demandent que chaque votants ne vote qu'une seule fois. L'autre outil majeur pour le cas non borné est Tamarin [90, 22], qui utilise la réécriture de multiensembles, mais qui, comme ProVerif, considère des propriétés de trace et la diff-équivalence. Tamarin n'est cependant pas entièrement automatique, mais plutôt interactif : il permet à l'utilisateur de guider l'outil. Ceci lui permet de prouver des cas difficiles à traiter automatiquement, mais bien sûr requiert plus de travail de l'utilisateur.

D'autre part, dans le cas de nombre bornés de sessions, plusieurs procédures ont été proposées récemment, pour *décider* l'équivalence de protocoles. L'un des premiers outils est SPEC [68], qui prouve une notion forte d'équivalence (la bisimulation) pour une signature de primitives cryptographiques fixée. Les outils plus récents prouvent une notion plus faible appelée l'équivalence de trace. APTE [45] décide l'équivalence de trace, également pour un ensemble de primitives fixé, mais supporte des constructions plus complexes telles que les branches else. Akiss [43] utilise des clauses de Horn, et supporte des primitives et théories équationnelles définies par l'utilisateur (parmi une large classe qui capture la plupart des primitives usuelles). Plus récemment, Deepsec [48] supporte, comme Akiss, des théories spécifiées par l'utilisateur et des constructions complexes (branches else, canaux privés). Il est bien plus efficaces que les outils précédents – et a été conçu pour établir une borne supérieur sur la complexité du problème de l'équivalence, dans une preuve que ce problème est coNEXP-complet. Un avantage d'Akiss et Deepsec est qu'ils tire partie du parallélisme pour accélérer la vérification. Enfin, SAT-Equiv [55] supporte une théorie fixée (contenant la plupart des primitives usuelles) et réduit l'équivalence à un problème de planification de graphe. Il est très efficace, mais impose certaines restrictions sur les protocoles considérés – il requiert une propriété de bon typage, et par conséquent il est parfois nécessaire d'ajouter des tags sur les messages pour encoder un protocole. Tous ces outils ont le problème qu'ils fonctionnent, d'une certaine façon, en considérant toutes les exécutions possibles des protocoles à vérifier, ce qui est nécessaire pour *décider* l'équivalence : ils ont donc tendance à souffrir, à des degrés divers, de problèmes d'efficacité lorsque le nombre de branches parallèles des processus est élevé.

Typage Les systèmes de types sont un outil courant dans le domaine des languages de programmation, et sont utilisés pour imposer statiquement des propriétés de sûreté telles que l'absence d'erreur à l'exécution des programmes. Une utilisation simple de types est de décrire la nature des valeurs considérées, avec des types tels que `int`, `string`, `bool`, etc. Les programmes sont alors vérifiés par un système de type contenant des règles telles que l'exemple (très simple) suivant, qui indique que le produit de deux entiers est aussi un entier :

$$\frac{x : \texttt{int} \qquad y : \texttt{int}}{x * y : \texttt{int}}$$

Les systèmes de types peuvent être vus comme une surapproximation des exécutions possibles des programmes, qui abstraient les valeurs par leur type. L'avantage d'une telle approche est qu'elle fournit des moyens efficaces (mais bien entendu incomplets) de prouver des propriétés qui seraient en général indécidables.

Les systèmes de types peuvent aussi être utilisés pour prouver des propriétés bien plus fortes dans le cadre de la sécurité. Ils ont été en particulier appliqués à la vérification de propriétés de trace telles que l'authentification ou la non-déductibilité [80, 38, 77]. Dans de tels cas, l'on peut par exemple marquer les valeurs qui ne doivent pas être publiées avec un type `secret`, et celles qui peuvent l'être sans risque avec un type `public`. La règle de typage simple suivante indiquerait alors que le chiffrement d'un message secret ne laisse fuir aucune information secrète (et est donc publiable) pourvu que la clef utilisée soit secrète :

$$\frac{x : \texttt{secret} \qquad k : \texttt{secret}}{\texttt{enc}(x, k) : \texttt{public}}$$

L'on peut alors vérifier le type d'un protocole entier pour s'assurer que tous les messages qui risquent d'être publiés ont bien pour type `public`.

Nous avons déjà évoqué F⋆ [96]. Tout comme son extension relationnelle rF⋆ [20], il repose sur un système de types riche avec des types dépendants et des raffinements : des types qui contiennent des formules logiques encodant des conditions complexes sur les valeurs auxquelles on les donne. Contrairement à l'approche décrite ci-dessus, son système de types n'est pas spécifiquement dédié aux protocoles de sécurité, mais permet d'écrire manuellement des formules logiques qui encodent des propriétés très complexes telles que des jeux cryptographiques. Le typage produit donc des obligations de preuve complexes, qui sont déchargées par F⋆ à un SAT-solver. De telles techniques ont été utilisées pour produire des implémentations vérifiées de protocoles complexes et de grande ampleur tels que TLS [72]. Cependant, un inconvénient de types si complexes est que, bien que la vérification de type elle-même soit automatique, un travail hautement non-trivial est requis de l'utilisateur, pour écrire les bons types qui permettront aux preuves de fonctionner.

7 Contributions

Cette thèse apporte plusieurs contributions au domaine de l'analyse formelle de protocoles de vote électronique, sur deux aspects principaux :

- d'une part, la vérification automatique de propriétés d'équivalence symboliques telles que le secret du vote, avec la conception d'un système de types pour l'équivalence;
- d'autre part, une étude des notions de secret du vote et de vérifiabilité, qui conduit au résultat surprenant que le secret du vote implique en fait la vérifiabilité individuelle, et à la mise au point d'une nouvelle définition par jeu du secret du vote, qui permet de considérer une urne malhonnête.

Un système de types pour l'équivalence symbolique Nous proposons une nouvelle approche au problème de la preuve automatique de propriétés d'équivalence, qui fait usage de systèmes de types. Comme expliqué précédemment, ils ont été appliqués au cas de propriétés de traces telles que l'accessibilité. La difficulté est que de telles techniques surapproximent l'ensemble des exécutions possibles du protocoles, en abstrayant les messages par leurs types. Cela permet d'assurer qu'un "mauvais" état (*e.g.* dans lequel l'attaquant apprend un secret) ne peuvent pas être atteint, en montrant que c'est le cas dans l'ensemble surapproximé des exécutions. Cette manière de faire ne serait cependant pas correcte pour l'équivalence. Même si les surapproximations de deux processus P et Q sont équivalentes, il se pourrait que certains comportements spécifiques de P ne soient pas possibles pour Q, ce qui briserait leur équivalence, mais pourrait ne pas être détecté à cause de la surapproximation. À la place, nous proposons un système de types qui peut vérifier deux processus P et Q ensemble pour s'assurer qu'ils ont les mêmes comportements, et nous utilisons les types pour établir des invariants sur les messages stockés dans des variables par ces processus. Durant le typage, notre système collecte certains sous-termes des messages émis dans ce que nous appelons une *contrainte*. Des tests supplémentaires sur cette contraintes assurent que non seulement P et Q ont les mêmes comportements, mais les messages observables lors d'une exécutions sont également indistinguables. Notre système de types supporte une signature de primitives cryptographiques fixée, qui inclut la plupart des primitives usuelles.

Nous prouvons que notre système de type implique correctement l'équivalence symbolique, même pour des processus comportant un mélange de parties répliquées (*i.e.* exécutées un nombre non borné de fois) et non-répliquées.

Nous avons également implémenté en collaboration avec Niklas Grimm un prototype de notre procédure de typage, que nous avons appliqué à plusieurs exemples de protocoles de la littérature (*e.g.* des protocoles d'échange de clef), pour comparer ses performances à celles des autres outils existants listés précédemment. Ces tests montrent que pour des nombres bornés de sessions, notre approche est très efficace, et souffre bien moins que les autres outils de grands nombres de sessions. Dans le cas non borné, nous supportons un mélange de processus répliqués et non répliqués. Notre surapproximation est plus précise que celle de ProVerif dans de tels cas, et sait utiliser le fait que certaines parties des processus ne sont exécutées qu'une seule fois. Cette flexibilité nous permet de prouver le secret du vote pour Helios, pour lequel ProVerif ne

sait pas conclure. En effet, l'attaque de Roenne précédemment décrite implique que la propriété n'est vraie que si les votants ne revotent pas, ce qui conduit ProVerif à trouver de fausses attaques. En échange, notre prototype requiert de légères annotations de types de la part de l'utilisateur, et est moins général en ce qui concerne la théorie équationnelle que ProVerif.

Secret du vote et vérifiabilité Nous étudions les définitions du secret du vote et de la vérifiabilité, et établissons que, étonnamment, le secret implique la vérifiabilité individuelle. Ce résultat, bien que contre-intuitif, ne contredit pas le résultat précédent de [49], car nous considérons des définitions établies avec un adversaire polynomial, plutôt que non borné. Expliquons maintenant l'intuition générale du résultat. Considérons un système de vote qui ne serait pas du tout vérifiable, où un adversaire pourrait d'une façon ou d'une autre empêcher les votes de n'importe quels votants de son choix d'être comptés, et ce sans être détectés. Dans un tel système, un adversaire souhaitant apprendre mon vote pourrait simplement bloquer tous ceux des autres votants : le résultat ne compterait alors que mon vote, et le révèlerait donc. Cette attaque ne fonctionnerait bien sûr que dans un exemple aussi extrême : nous montrons en fait comment la généraliser pour produire une attaque sur le secret du vote à partir de n'importe quelle attaque sur la vérifiabilité individuelle. Nous prouvons ce résultat à la fois dans des modèles symbolique et calculatoire, pour bien montrer qu'il ne s'agit pas d'un artefact lié au modèle, mais bien d'une propriété intrinsèque des notions de secret du vote et de vérifiabilité. Un aspect important de notre résultat, qui peut expliquer qu'il soit contre-intuitif, An important aspect of our result, that perhaps explains why it is counter-intuitive, est qu'il demande de considérer les mêmes hypothèses d'honnêteté ou malhonnêteté des autorités de l'élection pour le secret et la vérifiabilité. Dans les travaux existants en revanche, la vérifiabilité est généralement étudiée contre un attaquant contrôlant l'urne, alors que les notions de secret supposent implicitement une urne honnête.

Secret du vote contre une urne malhonnête Cette dernière remarque souligne une limitation des définitions (par jeu) existantes du secret du vote : elles supposent implicitement une urne honnête, alors que les systèmes de votent visent à protéger les votants d'une urne corrompue. Une observation similaire a été faite dans [67, 29]. Ceci nous conduit à la dernière contribution principale de cette thèse : la conception d'une définition de secret du vote contre une urne malhonnête, dans un modèle calculatoire. Notre définition est fondée sur BPRIV, une notion moderne de secret du vote contre une urne honnête, proposée dans [26]. Concevoir une définition lorsque l'urne est corrompue est rendu notoirement difficile par l'attaque décrite au paragraphe précédent. Un adversaire contrôlant l'urne peut toujours en retirer des bulletins, ce qui fait que le résultat donne plus d'information qu'il ne devrait sur les bulletins restants. Ceci viole trivialement la notion de secret naïve qui requiert qu'en toutes circonstances l'attaquant ne puisse obtenir aucune information sur les votes. Nous présentons un moyen de définir une notion légèrement plus faible, qui ne considère pas le comportement précédent comme une attaque, mais apporte tout de même des garanties significatives. Plus précisément, nous montrons que notre notion demeure suffisamment robuste, dans le sens où elle implique une notion de secret du vote par simulation. Cette simulation indique que le système de vote implémente de manière sûre une fonctionnalité idéale, qui rend explicite le pouvoir qu'un attaquant peut avoir contre ce système. Nous appliquons ensuite notre notion à plusieurs exemples de la littérature, Helios [10], Belenios [59] et Civitas [54], pour illustrer sa capacité à caractériser précisément le niveau de secret du vote qu'ils offrent.