UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Daniel DE ALMEIDA BRAGA

Cryptography in the Wild : The Security of Cryptographic Implementations

Thèse présentée et soutenue à IRISA, Rennes, le 14 Décembre 2022
Unité de recherche : UMR 6074

Rapporteurs avant soutenance :

Sébastien BARDIN	Chercheur Senior, CEA LIST, France
Kenneth PATERSON	Professeur, ETH Zürich, Suisse
Yuval YAROM	Associate Professor, Université d'Adélaïde, Australie

Composition du Jury :

Président :	Sandrine BLAZY	Professeure, Université de Rennes 1, France
Examinateur.rices :	Sylvain DUQUESNE	Professeur, Université de Rennes 1, France
	Clémentine MAURICE	Chargée de Recherche CNRS, Université de Lille, France
Rapporteurs :	Sébastien BARDIN	Chercheur Senior, CEA LIST, France
	Kenneth PATERSON	Professeur, ETH Zürich, Suisse
	Yuval YAROM	Associate Professor, Université d'Adélaïde, Australie
Dir. de thèse :	Pierre-Alain FOUQUE	Professeur, Université de Rennes 1, France
Encadrant :	Mohamed SABT	Maître de Conférence, Université de Rennes 1, France

Remerciements

[Only available on printed versions.]
[Disponible uniquement dans les versions imprimées.]

Contents

Remerciements **iii**

Résumé en Français **ix**
1. Contribution xii
2. Autres Contributions xvi
3. Artéfacts et Correctifs xviii

Publications **xxi**

Abbreviations and Notations **xxiii**

---

1. Introduction **3**
1.1. Contributions of this Thesis 5
1.2. Other Contributions 9
1.3. Artifacts and Patches 11
1.4. Outline 11

I. Preliminaries **15**

2. Modern CPU Microarchitectures and Related Attacks **17**
2.1. Optimizing the Execution 18
2.2. Memory Layout and Cache Hierarchy 19
2.3. Cache-Timing Attacks 23

3. Password Authenticated Key Exchange **29**
3.1. A Long List of Protocols 30
3.2. The Peculiar Case of SRP 31
3.3. The Wide Adoption of Dragonfly 34

4. Side Channel in PAKE implementations **39**
4.1. On the Danger of Leakage in PAKE 40
4.2. Assessing the Attack's Efficiency 41
4.3. Related Attacks on PAKEs 42

5. Related Topics and Threat Model **45**
5.1. Related Topics . . . 46
5.2. Threat Model . . . 47

II. Contributions **49**

6. Password Recovery Attack Against SRP Implementations **51**
6.1. Context and Motivation . . . 53
6.2. Attack on SRP Implementations . . . 54
6.3. Theoretical Amount of Leaked Information . . . 61
6.4. Experimentations . . . 63
6.5. Practical Impact . . . 68
6.6. Mitigations . . . 74

7. Remanent Side Channels in Dragonfly Implementations **77**
7.1. Context and Motivation . . . 79
7.2. Dragonblood Is Still Leaking . . . 81
7.3. Dragondoom: residual vulnerabilities in widespread implementations . . . 90
7.4. Comparative Analysis . . . 103
7.5. Discussion . . . 104

8. Usage and Usability of Constant-Time Verification Tools **107**
8.1. Introduction . . . 109
8.2. Background & Related Work . . . 110
8.3. Methodology . . . 116
8.4. Results . . . 121
8.5. Discussion . . . 133

9. Dragonstar: Formally Verified Cryptography for Dragonfly **137**
9.1. Context and Motivations . . . 139
9.2. Formally Enforcing Secret Independence . . . 140
9.3. Dragonstar: Formally Verified Cryptography for Dragonfly . . . 142
9.4. Discussions . . . 145

III. Conclusion and Future Work **147**

Conclusion **149**

Perspectives and Current Work **153**

Bibliography **155**

Appendix **179**

A. Chosen Group Attack Against ProtonMail' SRP **179**

B. Additional material regarding Dragondoom **181**
B.1. Results of the PDA on Affected Libraries 181
B.2. Experiments on BN_bin2bn 183

C. Survey **185**
C.1. Background 185
C.2. Library / Primitive 185
C.3. Tool use 187
C.4. Tool use: Dynamic instrumentation based 188
C.5. Tool use: Statistical runtime tests 189
C.6. Tool use: Formal analysis 189
C.7. Miscellaneous 189

Résumé en Français

Compte tenu de l'omniprésence des technologies numériques, il n'est pas surprenant de constater un accroissement de cyber-attaques. Alors que les prémisses de l'informatique ne bénéficiaient que de peu de sécurité, les systèmes modernes doivent considérer des modèles de menaces plus forts, et la sécurité est devenue une préoccupation de la plus haute importance. Cette problématique doit être abordée à plusieurs niveaux. L'un d'entre eux est axé sur la sécurité des données et des communications afin de fournir des garanties telles que la confidentialité, l'intégrité et l'authenticité. Ces propriétés, entre autres, peuvent être assurées par la cryptographie. La cryptographie est une discipline bien plus ancienne que l'informatique, qui est devenue une pièce maîtresse de la cybersécurité. Les premiers protocoles cryptographiques reposaient sur le secret de l'algorithme cryptographique (on peut penser au tristement célèbre chiffrement de César), mais depuis, le principe de Kerckhoff a façonné l'évolution du domaine :

> *"La conception d'un système ne devrait pas exiger le secret, et la compromission du système ne devrait pas incommoder les correspondants."*

Afin de respecter cette contrainte, les cryptosystèmes *modernes* sont basés sur un modèle mathématique. Le coeur de l'algorithme repose sur la difficulté d'un problème mathématique complexe qui est supposé être difficile (c'est à dire impossible avec la puissance de calcul actuelle) à résoudre sans connaître un élément secret, appelé la clé. L'une des contraintes de la cryptographie appliquée est que l'on doit tenir compte de la complexité globale de l'algorithme afin qu'il puisse être utilisé quotidiennement sans nuire à l'expérience de l'utilisateur. Dans la pratique, de multiples algorithmes (appelés *primitives*) sont combinés pour construire des protocoles, qui sont utilisés par des machines pour atteindre des objectifs particuliers, comme établir d'un canal de communication sécurisé. Il va sans dire que le secret de la clé est essentiel à la sécurité d'un algorithme cryptographique, et que sa compromission peut affecter l'ensemble du protocole.

Si la complexité du problème mathématique sous-jacent est nécessaire pour assurer la sécurité, elle ne suffit pas à rendre une primitive *sûre*. En pratique, les modèles mathématiques ne se traduisent pas parfaitement dans l'implémentation pour de multiples raisons. Cela fait de l'ingénierie cryptographique une partie essentielle et complexe de la sécurité des logiciels.

Lors de la mise en oeuvre d'un protocole cryptographique, les développeurs doivent tenir compte de ses performances tout en préservant sa sécurité. Cette tâche devient ardue lorsque certaines interactions spécifiques à divers niveaux d'implémentation peuvent entraîner des fuites d'informations internes pendant le calcul, ouvrant la voie à de nouvelles attaques non prises en compte dans le modèle théorique. Tout d'abord, au niveau logique, il est important d'être attentif à la validité des données traitées à chaque étape de du protocole. Les attaquants peuvent abuser de l'état du protocole en fournissant des données inattendues pour déduire des

informations du résultat ou en rejouant des données valides pour contourner une vérification. Ensuite, toute vulnérabilité logicielle générique (par exemple, liée à la gestion de la mémoire) peut entraîner une fuite de données arbitraires en mémoire (par exemple, une clé secrète), comme l'a démontré la tristement célèbre attaque Heartbleed sur OpenSSL [Gru14]. Enfin, les attaquants peuvent déduire des informations sur une valeur secrète en observant divers indicateurs. Pour illustrer ce dernier point, nous pourriont imaginer la façon dont les voleurs peuvent deviner la combinaison d'un coffre-fort simplement en écoutant les sons qu'il émet sur les bons chiffres. Ce processus de fuite indirecte d'informations sur un secret est connu sous le nom de fuite par *canal auxiliaire* et sera l'un des principaux sujets de cette thèse.

Depuis les travaux précurseurs de Kocher en 1996 [Koc96], les attaques par canal auxiliaire sont devenues un moyen répandu de contourner les hypothèses de sécurité, car les fuites permettent aux attaquants de réduire la complexité du problème sous-jacent, voire de le rendre trivial. Cette classe d'attaques peut prendre de multiples formes allant de l'interprétation du code d'erreur à la surveillance de la consommation d'énergie [KJJ99], de l'émanation électromagnétique (EM) [RR01] ou du temps de traitement [Koc96]. La plupart de ces fuites ont une racine commune : avec la complexité croissante des ordinateurs modernes, chaque cycle de traitement compte, et les composants sont souvent considérés comme isolés les uns des autres pour faciliter la construction, la maintenabilité et l'optimisation. Au fur et à mesure que la complexité augmente, il devient plus difficile de maintenir une abstraction parfaite d'un composant à l'autre. En particulier, de nombreuses optimisations modernes dépendent des données à traiter. Par conséquent, plus les systèmes sont complexes, plus ils sont sujets aux fuites.

Les sources les plus courantes de fuites par canaux auxiliaires dans les implémentations cryptographiques proviennent d'un flot de contrôle dépendant du secret (par exemple lorsqu'une implémentation effectue un branchement sur une valeur dérivée d'un secret), ou d'un accès mémoire dépendant du secret (par exemple lorsqu'un tableau est indexé avec une valeur secrète). Ces comportements entraînent une différence de temps dans le traitement, que ce soit sur l'opération globale ou sur les manipulations internes. Par conséquent, il en résulte une *attaque temporelle*.

De multiples aspects rendent les attaques temporelles spéciales par rapport à d'autres attaques par canal auxiliaire telles que l'analyse de consommation ou les attaques EM. Tout d'abord, elles peuvent être exécutées *à distance*, à la fois dans le sens d'exécuter du code en parallèle au code de la victime sans avoir besoin d'un accès local à l'ordinateur cible, mais aussi dans le sens d'interagir uniquement avec un serveur sur le réseau et de mesurer les temps de réseau [BB03] ou sur le Cloud [ZJRR12]. Par conséquent, contrairement à de nombreuses autres attaques par canal auxiliaire, les attaques temporelles ne peuvent pas être empêchées en limitant l'accès physique à la machine cible. Deuxièmement, les attaques temporelles ne laissent pas de traces sur la machine de la victime au-delà des journaux d'accès éventuellement suspects, et nous ne pouvons pas déterminer dans quelle mesure elles sont utilisées dans le monde réel, par exemple par des organismes gouvernementaux : les victimes ne sont pas en mesure de détecter de manière fiable qu'elles sont attaquées et l'attaquant ne le révelera jamais. Malheureusement, de telles vulnérabilités affectent encore les implémentations et les bibliothèques cryptographiques.

Une classe particulière d'attaques temporelles tire parti du comportement spécifique de ressources partagées génériques telles que processeur, chargée de traiter toute donnée ou instruction pendant l'exécution d'un programme, pour obtenir des informations basées sur la manière dont certaines données sont traitées. À partir de ces observations, les attaquants

peuvent deviner quelles instructions sont exécutées ou quelles données sont chargées par le processeur, donnant ainsi une granularité plus fine aux attaques temporelles. Cette classe d'attaques, appelée *attaques microarchitecturales*, est un outils très utile permettant aux attaquants d'extraire des informations. De plus, comme les cibles sont des ressources partagées génériques, les attaquants peuvent réaliser certaines d'entre elles au niveau du logiciel avec un processus non privilégié.

L'histoire a prouvé que sécuriser des implémentations représente un réel défi et que la moindre erreur suffit à compromettre l'ensemble des efforts investits.Ainsi, les protocoles standards sont souvent implémentés dans quelques bibliothèques de référence, qui sont ensuite utilisées comme dépendances tierces dans de nombreux autres projets. De cette façon, la communauté peut concentrer les efforts d'ingénierie et bénéficier d'une implémentation bien étudiée, sûre et efficace. En revanche, il s'agit d'une approche à double tranchant : lorsqu'une vulnérabilité touche cette implémentation, tous les projets qui en dépendent peuvent être affectés, ce qui entraîne une attaque à grande échelle à partir d'une seule vulnérabilité.

Une famille spécifique de protocoles s'est révélée particulièrement sensible à toute forme de fuite en raison de ses hypothèses de sécurité particulières : les protocole d'échange de clé authentifié par mot de passe (PAKE). Dans sa forme la plus simple, un protocole PAKE (ou PAKE en abrégé) permet à deux entités ne partageant rien d'autre qu'une chaîne à faible entropie (par exemple, un mot de passe) de s'authentifier mutuellement tout en établissant une session sécurisée. Ainsi, les PAKEs suppriment le besoin d'une infrastructure à clé publique. Compte tenu de l'omniprésence des mots de passe comme méthode d'authentification, les PAKEs sont intéressants pour de nombreuses solutions et de tels protocoles ont été déployés à grande échelle au cours des dernières décennies. Une illustration récente de leur popularité est la standardisation d'un PAKE dans la nouvelle norme *Wi-Fi Protected Access* 3 (WPA3), laissant présager un déploiement sur des milliards de dispositifs dans les années à venir [All21].

Cependant, comme la sécurité du protocole repose désormais sur la connaissance d'une chaîne à faible entropie, toute fuite sur une valeur liée au mot de passe peut compromettre l'authenticité et la confidentialité du canal de communication. Cette propriété les rend particulièrement vulnérables à toute attaque par canal auxiliaire, comme nous le démontrerons dans nos contributions.

Alors que les fuites par canaux auxiliaires continuent d'affecter les applications les plus répandues, le besoin de contre-mesures appropriées ne cesse de croître. Les chercheurs ont proposé diverses stratégies pour corriger et réduire la propagation des canaux auxiliaires. En particulier, si l'on considère les attaques microarchitecturales basées sur le logiciel, les mesures de lutte peuvent être déployées à plusieurs niveaux : sur la couche de l'espace utilisateur, la couche système ou la couche matérielle. La plupart des solutions génériques visent à limiter le partage des ressources ou de masquer leur accès. L'isolation des ressources ou la randomisation de l'accès à ce niveau est un défi, mais de nombreuses solutions existent au niveau du système ou du matériel. Nous renvoyons le lecteur à l'étude [LZJZ21], sections 5.1 et 5.2, pour de plus amples informations sur l'état de l'art de ces mécanismes. Ces mesures d'atténuation constitueraient une solution générique à ces attaques microarchitecturales. Cependant, les solutions proposées sont difficiles à déployer (par exemple, des modifications matérielles) et peuvent entraîner une dégradation significative des performances du système. Un moyen plus abordable est de viser un correctif au niveau de l'application, empêchant les attaques temporelles les plus courantes en ciblant les implémentations dépendantes du secret. Cela peut être réalisé en appliquant une discipline de développement appelée *indépendance vis à vis du secret* : toutes les branches et tous les accès à la mémoire ne doivent dépendre que

des valeurs publiques. Cependant, cette propriété peut être difficile à garantir, surtout si l'on considère des bases de code importantes et complexes. Compte tenu de la complexité des bibliothèques modernes, la vérification assistée par ordinateur est un moyen prometteur d'éradiquer la dépendance secrète [BBB+21].

1. Contribution

Dans cette thèse, nous avons étudié plusieurs implémentations cryptographiques et avons remarqué la récurrence de certains types de vulnérabilités. Notre recherche se concentre principalement sur l'analyse des implémentations déployées de PAKE, car elles sont particulièrement vulnérables à toute forme d'attaque par canal auxiliaire. Pour démontrer la portée pratique des attaques que nous avons découvertes, nous avons fourni une implémentation de la Preuve de Concept (PdC) en tirant parti d'attaques microarchitecturales. Compte tenu de ces vulnérabilités et de leur portée, nous avons étudié leur origine et les solutions actuelles pour s'en prémunir. Dans nos principales contributions, nous répondons aux questions de recherche suivantes :

I. Les protocoles PAKE et leurs implémentations comprennent-ils des protections spécifiques contre les canaux auxiliaires, compte tenu de leurs hypothèses particulières ?

II. Pourquoi les logiciels cryptographiques contemporaines ne sont pas exempts de vulnérabilités en matière d'attaques temporelles ?

III. Comment pouvons-nous fournir des garanties contre les attaques par canaux auxiliaires au niveau du logiciel ?

Les contributions suivantes sont divisées en trois catégories. Premièrement, nous avons analysé deux PAKEs largement déployés : Secure Remote Password protocol (SRP) [Wu09], un protocole encore déployé pour des raisons historiques, et Dragonfly [Har08], un PAKE moderne utilisé dans WPA3. Dans les deux cas, nous avons découvert et exploité des fuites d'information permettant de récupérer le mot de passe, ce qui indique un manque de sensibilisation à l'aspect pratique de telles attaques. Deuxièmement, pour comprendre la persistance de ces fuites, nous avons étudié l'utilisation des outils de vérification assistée par ordinateur par les ingénieurs en cryptographie. Cela a révélé une faible utilisation des outils d'analyse automatique du critère "temps constant" dans la communauté, malgré la prolifération de contributions académiques. En recueillant les commentaires des développeurs, nous avons identifié des déficience dans le déploiement des outils et établi des recommandations. Enfin, comme nous n'avons pas trouvé d'outil de vérification formelle passant à l'échelle d'implémentation complexes, nous avons exploré une autre façon de fournir une implémentation dont l'exécution est indépendante du secret, en générant du code C formellement vérifié. Nous avons développé un module formellement vérifié, à intégrer dans un daemon Wi-Fi répandu, garantissant des contre-mesures durables pour les vulnérabilités que nous avons découvertes dans les implémentations de Dragonfly.

Analyse d'Implémentations de PAKE Largement Déployés

Le concept de PAKE n'est pas nouveau : il a été initialement décrit par Bellovin et Merritt en 1992 [BM92] et a reçu une attention considérable depuis lors. Cependant, la plupart de la

littérature existante se concentre uniquement sur la conception abstraite de tels protocoles, laissant de côté leurs implémentations de référence ou déployées. Ceci est regrettable, d'autant plus que les mots de passe sont particulièrement vulnérables à toute fuite de secret en raison de leur faible entropie. En outre, la RFC 8125 [Sch17] exige explicitement que tout PAKE soit protégé contre les attaques par de canal auxiliaire. Dans cette thèse, nous visons à combler cette lacune en examinant les différentes implémentations de deux PAKE largement déployés : (i) SRP, et (ii) l'instanciation de Dragonfly dans Simultaneous Authentication of Equals (SAE) utilisé dans WPA3. En étudiant à la fois un ancien et un nouveau PAKE, nous cherchons à répondre à la question de recherche **I**. Nous espérons que ces contributions sensibiliseront au besoin d'algorithmes et d'implémentations en temps constant (formellement vérifiés) qui ne dépendent pas de correctifs incrémentiels pour garantir la sécurité jusqu'à ce que la prochaine fuite soit trouvée.

Le Cas d'un PAKE Legacy : SRP

SRP est sans doute parmis les PAKEs les plus largement utilisés. Il tire sa popularité de sa spécification libre de tout brevet, de sa conception assez simple et de la disponibilité d'implémentations open-source efficaces. Dans cette contribution, nous montrons que de nombreuses implémentations de SRP ne résistent pas aux attaques par dictionnaire hors ligne. En effet, nous identifions d'abord certains vecteurs de fuite du côté client de l'implémentation de SRP fournies par OpenSSL. Ensuite, nous l'exploitons par des attaques par canaux auxiliaire sur le cache du processeur. Nous simplifions l'exécution de notre attaque en automatisant l'ensemble du processus, y compris l'espionnage des instructions exécutées, l'interprétation des mesures de cache et la récupération du mot de passe. Enfin, nous montrons comment notre attaque s'applique à des solutions largement déployées, telles que Proton Mail et Apple HomeKit. Pour chaque projet, nous indiquons le composant vulnérable et les implications en matière de sécurité. Nous avons notifié les projets impactés et fourni un soutien pour l'intégration des correctifs. Au total, sept patchs ont été développés et publiés suite à nos travaux. Enfin, nous mettons à disposition un PdC sur différents projets et les rendons open-source.

> **Reference:** Ces travaux sont issus d'une collaboration avec Mohamed Sabt et Pierre-Alain Fouque. Ils ont été publiés dans les proceedings de *CCS '21 : 2021 ACM SIGSAC Conference on Computer and Communications Security* [BFS21].

Le Cas d'un PAKE Moderne : Dragonfly

En 2018, la Wi-Fi Alliance a annoncé WPA3-SAE pour remplacer l'authentification par clé pré-partagée définie par WPA2, qui a fait l'objet de nombreuses attaques, notamment des KRACKs [VP17]. SAE est défini dans la norme IEEE 802.11-2016 [IEE21], et met en oeuvre une légère variante de la RFC Dragonfly définie dans [Har15]. Peu après son introduction, Vanhoef et Ronen ont présenté *Dragonblood*, un ensemble de vulnérabilités dans les implémentations de WPA3, en particulier contre sa méthode de conversion de mot de passe [VR20]. En réponse, la Wi-Fi Alliance a publié des conseils d'implémentation afin éviter les attaques.

Dragonblood fuit encore. Nous avons identifié plusieurs implémentations dans lesquelles certaiens instructions ne sont exécutées que lorsque le mot de passe est correctement convertit. Nous montrons comment un attaquant peut utiliser des attaques sur le cache du CPU pour faire fuir de l'information sur le mot de passe. Notre attaque s'appuie sur des travaux antérieurs et améliore considérablement les performances de l'attaque originale (*Dragonblood*), en extrayant plus d'informations avec moins de mesures. Nous appliquons nos résultats au daemon sans fil iNet Wireless Daemon (iwd)[1], d'Intel, à sa bibliothèque cryptographique connexe ell[2] et à FreeRADIUS[3]. Nous avons non seulement communiqué les vulnérabilités identifiées aux mainteneurs de ces projets open-source, mais nous les avons également aidés à corriger le code vulnérable. Au total, trois correctifs ont été développés et publiés. Cette contribution montre que malgré le respect des directives de la Wi-Fi Alliance, les daemons Wi-Fi peuvent toujours être vulnérables aux attaques par canal auxiliaire basées sur le cache.

> **Reference:** Ces travaux sont issus d'une collaboration avec Mohamed Sabt et Pierre-Alain Fouque. Ils ont été publiés dans les proceedings de *ACSAC '20 : Annual Computer Security Applications Conference* [BFS20a].

Dragondoom : Vulnérabilités Résiduelles Provenant de Dépendances. Après de nombreux efforts, et quatre ans après la standardisation de WPA3 et la modification du protocole original, il est raisonnable de supposer que les vecteurs d'attaque présentés dans [VR20 ; BFS20a] ne sont plus pertinents. En effet, on s'attend à ce que les principales implémentations open-source de WPA3 (par exemple, hostap) soient exemptes d'attaques par canaux auxiliaires microarchitecturaux, d'autant plus qu'elles ont été analysées à la fois manuellement et en utilisant des outils automatiques pour détecter les fuites microarchitecturales, tels que MicroWalk [WMES18a]. Nous remettons en question cette croyance et soulevons à nouveau la question de la sécurité pratique de Dragonfly. Nous ne nous contentons pas de signaler une autre attaque microarchitecturale contre Dragonfly. Nous mettons en lumière une source de fuite inexploitée qui affecte toujours les implémentations WPA3. En effet, nous montrons que les daemons largement déployés sont ancore affectés par des fuites d'information causées par les bibliothèques cryptographiques qu'ils supportent. En pratique, les implémentations Dragonfly s'appuient principalement sur des bibliothèques tierces pour effectuer les opérations cryptographiques requises. Ces bibliothèques ne fournissent pas toujours des implémentations indépendantes du secret pour leurs fonctions. Malheureusement, les impacts sur la sécurité de l'appel d'une fonction externe dans Dragonfly n'ont pas été étudiés. Dans ce travail, nous analysons et suivons les valeurs dépendant des mots de passe dans l'opération de conversion de mots de passe de Dragonfly, en particulier dans les appels à des fonctions externes. Nos analyses ont été enrichissantes : nous avons remarqué que la plupart des bibliothèques que nous avons évaluées sont vulnérables aux attaques par canal auxiliaires, faisant fuire de l'information sur des valeurs liées au mot de passe. Contrairement aux travaux précédents, nous montrons que les vulnérabilités ne concernent pas seulement l'ancienne méthode de "hunting-and-pecking", mais aussi la méthode basée sur l'algorithme *Simplified Shallue-van de Woestijne-Ulas* (SSWU), récemment implémentée, qui est censée être indépendante du secret par design.

[1] https://iwd.wiki.kernel.org/
[2] https://git.kernel.org/pub/scm/libs/ell/ell.git/
[3] https://freeradius.org/

Nous exposons nos vulnérabilités dans plusieurs bibliothèques cryptographiques utilisées par des implémentations SAE répandues : hostap, iwd et FreeRADIUS. Nous décrivons comment la fuite peut conduire à une attaque par dictionnaire hors ligne, ce qui va à l'encontre de l'objectif de l'utilisation d'un protocole PAKE dans WPA3. À cette fin, nous utilisons des techniques microarchitecturales connues pour surveiller les instructions ciblées. Nous démontrons l'efficacité de notre attaque en récupérant avec succès les mots de passe des utilisateurs dans la dernière version de hostap en utilisant OpenSSL, qui est le paramètre par défaut du daemon Wi-Fi sur la plupart des systèmes basés sur Linux. Nous fournissons également des indications suggérant que d'autres implémentations, telles que iwd/ell, sont également affectées par cette vulnérabilité à un degré plus élevé.

Reference: Ces travaux sont issus d'une collaboration avec Mohamed Sabt et Pierre-Alain Fouque. Ils sont en cours de soumission et n'ont pas encore été présentés publiquement.

Enquête sur le Manque de Vérification Assistée par Ordinateur

Il est bien établi que la sécurité d'une primitive cryptographique ne dépend pas seulement de sa robustesse théorique. Les problèmes d'implémentation constituent une partie essentielle pour garantir que le code en cours d'exécution ne fait fuire aucune information secrète. Une classe de vulnérabilités qui nous intéresse est l'une des attaques par canal auxiliaire les plus répandues : *les attaques temporelles*. Comme nous l'avons détaillé précédemment, en principe, ces attaques ne sont pas si difficiles à atténuer. En effet, nous devons nous assurer que l'accès à la mémoire et le flux de contrôle sont indépendants des secrets. Au vu de l'évolution du nombre d'attaques découvertes, la recherche défensive a également évolué, et de nombreuses techniques existent pour vérifier automatiquement les implémentations, soutenues par une large gamme d'outils [BBB+21]. Ces outils diffèrent dans leurs objectifs et offrent diverses garanties. Pourtant, ils démontrent collectivement que l'analyse automatique de programmes pour garantir l'indépendance vis à vis du secret est possible.

Dans cette contribution, nous proposons de répondre à notre question de recherche **II**. À cette fin, nous menons une enquête en ligne à méthode mixte auprès de 44 développeurs de 27 bibliothèques cryptographiques populaires. Par le biais de cette enquête, nous recherchons l'origine de la persistance de ces attaques par canal auxiliaire. Sur la base de l'enquête, nous identifions les lacunes des outils d'analyse existants et émettons des recommandations à l'intention des développeurs d'outils. Nos résultats montrent que les défauts des outils implique que les tests d'indépendance vis à vis du secret ne figurent pas en tête de liste des tâches des développeurs. L'enquête nous a permis de comprendre que les attaques temporelles pourraient disparaître ou se raréfier avec des outils plus utilisables, nécessitant peu de travail et donnant des résultats faciles à comprendre.

Reference: Ces travaux sont issus d'une collaboration avec Jàn Jancar, Marcel Fourné, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, et Yasemin Acar. Ils ont été publiés dans les proceedings de *IEEE Symposium on Security and Privacy 2022* [JFB+22].

Implémentations Formellement Vérifiées : une Alternative à la Vérification

Dans les contributions précédentes, nous avons identifié deux problèmes sérieux qui ont un impact sur la sécurité des implémentations modernes. Premièrement, les attaques par canal auxiliaire sont effectivement pratiques. Deuxièmement, bien que riche, l'écosystème actuel des outils de vérification du critère temps-constant manque de maturité en termes d'utilisabilité, ce qui entrave le déploiement des outils parmi les ingénieurs en cryptographie.

Compte tenu des vulnérabilités résiduelles qui affectent la couche cryptographique des implémentations de Dragonfly, nous avons cherché à vérifier l'ensemble de son implémentation afin de nous assurer de l'absence d'autre fuite d'information. Dans l'ensemble, la vérification d'un sous-ensemble conséquent d'une bibliothèque est une tâche ardue, et nous n'avons pas trouvé d'outil passatn à l'échelle, tout en fournissant des garanties formelles. Cependant, même des outils plus simples, comme timecop [Nei ; Lan10], ont révélé de multiples fuites qui pourraient conduire à des attaques permettant la récupération de mots de passe. Ainsi, entre le manque de garanties et la tâche éreintante d'étudier toutes les dépendances vis à vis d'un secret, nous avons décidé d'aborder le problème différement, en répondant à notre question de recherche **III**.

Au lieu de vérifier une implémentation existante, une autre approche consiste à générer une implémentation formellement vérifiée à partir de spécifications bien définies. En considérant WPA3, le protocole lui-même peut être divisé en plusieurs couches, interagissant les unes avec les autres : la couche réseau, la couche protocole et la couche cryptographique. Nous fournissons une implémentation C formellement vérifiée de la couche cryptographique, garantissant l'indépendance du secret. Cette implémentation est conçue afin de pouvoir être intégrée dans l'un des daemon Wi-Fi les plus déployés : hostap. Nous avons construit ce plugin pour hostap, où tous les appels cryptographiques dans SAE redirigent vers la bibliothèque HACL* formellement vérifiée [ZBPB17]. Nous fournissons l'implémentation C, qui atteint nos objectifs de sécurité tout en gardant des performances raisonnables.

> **Reference:** Ces travaux sont issus d'une collaboration avec Mohamed Sabt, Pierre-Alain Fouque, Karthikeyan Bhargavan, et Natalia Kulatova. Ils poursuivent les précédents travaux sur Dragonfly [BFS20a]. Ils sont en cours de soumission et n'ont pas encore été présentés publiquement.

2. Autres Contributions

Cette section décrit deux contributions que nous avons exclues du manuscrit afin de maintenir une structure cohérente. Néanmoins, nous les décrivons ici afin de dresser un tableau complet de toutes les contributions.

Vulnérabilités dans le Standard SCP10

Les spécifications des cartes GlobalPlatform (GP) sont les normes de facto pour l'industrie des cartes à puce. Ces spécifications définissent, entre autres, la famille de Secure Channel Protocol (SCP) [Tec18]. Comme son nom l'indique, les canaux SCP empêchent l'observation ou la falsification de la communication avec la carte. SCP01, SCP02 et SCP03 sont des mécanismes de chiffrement à clés symétriques utilisant DES (utilisé dans SCP01), Triple DES à deux clés (dans SCP02) et AES (dans SCP03). GP définit également deux canaux

SCP reposant sur le chiffrement asymétrique : SCP10 et SCP11. Malgré leur importance, les protocoles SCP ont été longtemps négligés dans la littérature. À notre connaissance, Sabt *et al.* ont fourni la première cryptanalyse pour les protocoles SCP02 et SCP03 [ST16]. Ils ont prouvé que SCP03 fournit des garanties de sécurité plus fortes en empêchant l'Attaque par Substitution d'Algorithme (ASA) [BJK15]. Ils ont également montré une attaque théorique contre SCP02 due à une mauvaise utilisation du vecteur d'initialisation. Dans [AF18], Avoine *et al.* ont montré une attaque plus pratique contre SCP02, dans laquelle ils exploitent les oracles de remplissage CBC pour récupérer certaines données secrètes.

Nous avons étudié les spécifications de SCP10, qui devient plus pertinent à déployer étant donné la dépréciation de SCP02 et un écosystème de plus en plus ouvert. Notre analyse identifie divers défauts de conception dans SCP10 qui permettent des attaques pratiques. En effet, nous constatons que SCP10 chiffre les données sans remplissage aléatoire. De plus, nous notons qu'il réutilise la même paire de clés RSA pour le chiffrement et la signature. Nous tirons parti du fait que PKCS#1 v1.5 [MKJR16] est encore pour la signature afin de déchiffrer des messages sensibles en appliquant les attaques de Coppersmith [Cop96 ; Cop97] et de Bleichenbacher [Ble98 ; BFK+12] dans différents contextes. En utilisant de vraies cartes à puce, nous évaluons la complexité de nos attaques. Il est intéressant de noter qu'une attaque par oracle de remplissage provenant d'une différence de temps dans le traitement est susceptible d'affecter de nombreuses implémentations étant donné le manque de spécification, le remplissage peu commun et l'environnement contraint. Enfin, nous proposons une implémentation sécurisée des spécifications SCP10 qui est conforme à la dernière recommandation du National Institute of Standards and Technology (NIST) en termes d'échange de clé [BCR+19]. GlobalPlatform s'est montré réactif quant à notre analyse. En conséquence, nous avons dirigé un groupe de travail avec leurs partenaires pour atténuer les faiblesses identifiées. Ce processus a aboutit à une modification de SCP10 offrant une meilleure sécurité [Tec20].

Il serait intéressant d'étendre nos travaux à SCP11, qui repose sur la cryptographie basée sur les courbes elliptiques. Bien que SCP11 ne soit encore défini que comme un amendement, les travaux existants au sein de GP reflètent son adoption future prévisible pour les cartes SIM embarquées. Malheureusement, à ce jour, aucune analyse connue n'a examiné la sécurité de SCP11.

> **Reference:** Ces travaux sont issus d'une collaboration avec Mohamed Sabt et Pierre-Alain Fouque. Ils ont été publiés dans les proceedings de *IACR Transactions on Cryptographic Hardware and Embedded Systems 2020* [BFS20b].

2.1. Attaque par Choix de Groupe sur l'Authentification de ProtonMail

ProtonMail est une solution de messagerie qui a construit sa réputation autour de son investissement dans la sécurité. En effet, elle se distingue des autres solutions en fournissant un chiffrement automatique des courrier électroniques côté client (protégeant les utilisateurs d'un serveur malveillant) et en rendant son code consultable. Ce qui nous intéresse particulièrement, c'est que l'authentification de l'utilisateur repose sur SRP pour obtenir le jeton d'authentification, ce qui permet d'accéder à la gestion du compte et éventuellement (selon la version) aux courriels chiffrés. Nous nous sommes intéressés à une application particulière : leur client python[1].

[1] `https://github.com/ProtonMail/proton-python-client`

En abusant des comportements particuliers de deux modules compris dans leur dépendances (pySRP et python-gnupg), nous sommes parvenu à contourner la vérification des signatures sur les éléments de groupe pour la session SRP et fournir des paramètres personnalisés. En utilisant des paramètres judicieusement choisis, les attaquants peuvent résoudre le problème du logarithme discret, récupérer la clé éphémère de l'utilisateur et effectuer des attaques par partitionnement de dictionnaire sur SRP.

Compte tenu des détails d'implémentation, le modèle de menace considère soit un serveur malveillant, soit des attaquants capables de contourner l'épinglage des certificats (qui n'est disponible que pour ProtonVPN au moment du test) et d'avoir leur clé publique ssh dans le trousseau de clés de la victime.

L'annexe A contient plus de détails sur l'attaque.

3. Artéfacts et Correctifs

Parallèlement à nos diverses contributions, nous attachons une importance particulière à la reproductibilité de nos résultats. Dans ce sens, nous fournissons des artéfacts dans des dépôts séparés, principalement en tant que PdC. En outre, nous avons fourni des correctifs ou un retour d'information sur les correctfs apportés aux projets affectés par nos attaques. Nous énumérons ci-dessous les contributions pratiques qui découlent de notre travail.

SCP10. Précedemment, nous avons décrit deux attaques sur l'établissement de session SCP10 sur une carte à puce. Nous avons implémenté les deux et divulgué publiquement les applets utilisés pour monter l'attaque de l'oracle de remplissage et l'attaque de récupération de clé. En plus de l'applet vulnérable, nous avons implémenté les contre-mesures suggérées pour avoir une idée pratique de la surcharge de performance qu'elles entraînent. Toutes les applets sont disponibles à l'adresse `https://github.com/ddealmei/SCP10-attack`.

Les problèmes que nous avons révélés ont conduit à une modification de la norme.

Dragonfly. Pour les deux contributions révélant des attaques sur Dragonfly, nous fournissons toutes les données et les scripts d'attaque avec un Dockerfile. Cette configuration permet de reproduire facilement l'ensemble de la chaine d'attaque, de l'agrégation des fuites à la récupération des mots de passe. Nous fournissons également notre jeu de données de fuites d'informations.

- La première attaque, démontrée sur iwd/ell, est disponible à l'adresse `https://gitlab.inria.fr/ddealmei/poc-iwd-acsac2020`.
- La seconde attaque, démontrée sur hostap et OpenSSL, est disponible à l'adresse `https://gitlab.inria.fr/ddealmei/artifact_dragondoom`.
- Un fournisseur cryptographique formellement vérifié, avec les opérations nécessaires pour implémenter Dragonfly, est disponible en tant que plugin pour hostap. Le dépôt contient un fichier Dokerfile pour éviter les problèmes de dépendance et faciliter la construction. Il contient des données de test (fuzzing différentiel avec OpenSSL) et des scripts d'évaluation. L'artéfact est disponible à l'adresse `https://gitlab.inria.fr/ddealmei/artifact_dragonstar`.

De plus, plusieurs correctifs et avis de sécurité ont été publiés par hostap (voir l'avis de sécurité), iwd/ell (voir les fils de discussion [1], [2], [3] et [4]) FreeRADIUS (voir le journal des modifications et les commits [1] et [2]) pour corriger ces vulnérabilités.

SRP. Pour la contribution sur SRP, l'artefact de la PdC exploitant un canal axiliaire dans l'implémentation de SRP par OpenSSL est disponible à l'adresse `https://gitlab.inria.fr/ddealmei/poc-openssl-srp`. L'attaque est appliquée sur OpenSSL, pySRP et Apple HomeKit pour démontrer sa portée. Après la divulgation, nous avons contacté tous les projets affectés que nous avons pu trouver et les avons soutenus dans leur processus de correction en soumettant des correctifs (sur OpenSSL, Apple HomeKit, pySRP, le client python de ProtonMail, node-bignum, ruby/openssl et Erlang OTP).

Publications

In this chapter, we list peer-reviewed publications in the proceedings of international conferences in chronological order. This thesis is built on the basis of these publications and some ongoing work which have not been peer-reviewed yet.

Publication I [BFS20b] *The Long and Winding Path to Secure Implementation of GlobalPlatform SCP10*, Daniel De Almeida Braga, Pierre-Alain Fouque and Mohamed Sabt. Published in the proceedings of TCHES 2020, Issue 3 (p.196–218)

Publication II [BFS20a] *Dragonblood is Still Leaking: Practical Cache-based Side Channel in the Wild*, Daniel De Almeida Braga, Pierre-Alain Fouque and Mohamed Sabt. Published in the proceedings of ACSAC 2020 (p.291-303)

Publication III [BFS21] *PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild*, Daniel De Almeida Braga, Pierre-Alain Fouque and Mohamed Sabt. Published in the proceedings of CCS 2021 (p.2497-2512)

Publication IV [JFB+22] *"They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks*, Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, Yasemin Acar. Published in the proceedings of IEEE S&P 2022 (p.755-772)

Abbreviations and Notations

AP Access Point.

CFRG Crypto Forum Research Group.

CPU Central Processing Unit.

DLP Discrete Logarithm Problem.

EAP Extensible Authentication Protocol.

ECC Elliptic Curve Cryptography.

ECDSA Elliptic Curve Digital Signature Algorithm.

HMAC Hash-based Message Authentication Code.

iwd iNet Wireless Daemon.

KDF Key Derivation Function.

L1 first-level cache.

L1d data cache.

L2 second-level cache.

LLC Last Level Cache.

LSB Least Significant Bit.

MAC Media Access Control.

MitM Machine-in-the-Middle.

MSB Most Significant Bit.

OS Operating System.

PAKE Password-Authenticated Key Exchange protocols.

PDA Performance Degradation Attack.

PoC Proof of Concept.

RAM Random Access Memory.

SAE Simultaneous Authentication of Equals.

SCA Side-Channel Attack.

SMT Satisfiability Modulo Theories.

SRP Secure Remote Password protocol.

SSID Service Set Identifier.

SSWU Simplified Shallue-van de Woestijne-Ulas.

TLS Transport Layer Security.

WPA Wi-Fi Protected Access.

Introduction

1 Introduction

Considering the ubiquitous usage of digital technologies, it is unsurprising to notice an increasing cyber-attack rate. While the premise of informatics had little to no security, modern systems have to consider stronger threat models, and security has become of concern of the utmost importance. This concern needs to be addressed at multiple levels. Some of them are focused on the security of data and communications to provide guarantees such as confidentiality, integrity, and authenticity. These properties, among others, can be ensured using cryptography.

Cryptography is a discipline far older than informatics, which has become a central piece of cyber-security. First cryptographic protocols relied on the secrecy of the cryptographic algorithm (one can think of the infamous Caesar cipher), but since then Kerckhoffs principle shaped the evolution of the field:

> *"The design of a system should not require secrecy, and compromise of the system should not inconvenience the correspondents."*

To follow this constraint, *modern* cryptosystems are based on a mathematical model. The core algorithm relies on the hardness of a complex mathematical problem which is believed to be difficult (read impossible with current computational power) to solve without knowing a secret element, referred to as the key. One constraint of practical cryptography is that one must consider the overall complexity of the algorithm so that it may be used every day without impeding the user experience. In practice, multiple algorithms (called *primitives*) are combined to build protocols, which are used by machines to achieve particular goals, such as establishing a secure communication channel. Needless to say that the secrecy of the key is essential to the security of a cryptographic algorithm, and compromising it may affect the entire protocol.

While the hardness of the underlying mathematical problem is necessary to provide security, it is not enough to make a primitive secure. In practice, the mathematical models do not perfectly translate to the implementation for multiple reasons. This makes cryptographic engineering an essential and complex part of software security.

When implementing a cryptographic protocol, developers must account for its performance while preserving its security. This task becomes daunting when some specific interactions at various implementation levels may leak internal information during the computation, opening to new attacks not considered in the theoretical model. First, on a logical level, protocols must be very careful about the data they are processing at each step. Attackers may abuse the protocol's state by providing unexpected data to infer information from the result or by replaying valid data to bypass a verification. Second, any generic software vulnerability (*e.g.,* related to memory safety) may leak arbitrary data in memory (*e.g.,* a secret key), as demonstrated by the infamous Heartbleed attack on OpenSSL [Gru14]. Finally, attackers may infer information on a secret value by observing various indicators. An illustrative example

would be how robbers may guess a safe combination simply by listening to how it clicks on the correct numbers. This process of indirectly leaking information on a secret is known as *side-channel* and will be one of the main topics of this thesis.

Since the seminal work of Kocher in 1996 [Koc96], side-channel attacks have become a prominent way of defeating security assumptions, as the leakages allow attackers to lessen the complexity of the underlying problem or even make it trivial. This class of attacks may take multiple forms, from interpreting error code to monitoring power consumption [KJJ99], electromagnetic (EM) emanation [RR01] or processing time [Koc96]. Most of them share a common root: with the increasing complexity of modern computers, each processing cycle counts, and components are often considered isolated from each other to ease building, maintainability, and optimization. As complexity increases, keeping a perfect abstraction from one component to another becomes harder. In particular, many modern optimizations depend on the data to be processed. Hence, the more complex systems get, the more prone to leakage they become.

The most common sources of side-channel leaks in cryptographic code are secret-dependent execution *e.g.,* when an implementation branches on a value derived from a secret, or performs secret-dependent memory access (*e.g.,* when an array is indexed with a secret value). These behaviors result in a timing difference in the processing, whether it is on the overall operation or on internal manipulations. Hence, it results on a *timing attack*.

Multiple aspects make timing attacks special compared to other side-channel attacks such as power-analysis or EM attacks. First, they can be carried out *remotely*, both in the sense of running code in parallel to the victim code without the need for local access to the target computer, but also in the sense of only interacting with a server over the network and measuring network timings [BB03] or over the Cloud [ZJRR12]. As a consequence, unlike many other side-channel attacks, timing attacks cannot be prevented by restricting physical access to the target machine. Second, timing attacks do not leave traces on the victim's machine beyond possibly suspicious access logs, and we do not know at all to what extent they are being carried out in the real world, for example by government agencies: victims are not able to reliably detect that they are under attack and the attacker will never tell. Unfortunately, such vulnerabilities still plague cryptographic implementations and libraries.

A particular class of timing attacks takes advantage of the specific behavior of generic shared resources such as the Central Processing Unit (CPU), in charge of processing any data or instruction processed during the execution of a program, to get information based on how some data is processed. From these observations, attackers can guess which instructions are executed or which data is loaded by the CPU, bringing the timing attacks to a finer granularity. This class of attacks, called *microarchitectural attacks*, is a very convenient way for attackers to extract information. Moreover, as the targets are generic shared resources, attackers can perform some of them from the software level with an unprivileged process.

History has proven that securing practical implementations is challenging and that the slightest mistake can lead to disastrous security consequences. Namely, standardized protocols tend to be implemented in a few reference libraries, which are then used as third-party dependencies in many other projects. This way, the community can focus the engineering effort in one place and benefit from a well-studied, secure and efficient implementation. This is a double-edged approach: when a vulnerability impacts this implementation, every project relying on it may be affected, resulting in a large-scale attack from a single vulnerability.

A specific family of protocols has been revealed to be particularly sensitive to any form of leakage due to their particular security assumptions: Password-Authenticated Key Exchange

protocols (PAKE). In its simplest form, a PAKE protocol (or PAKE for short) allows two parties sharing nothing but a low-entropy string (*e.g.,* a password) to authenticate each other while establishing a secure session. Thus, PAKEs remove the need for Public-Key Infrastructure (PKI). Considering the ubiquity of passwords as an authentication method, PAKEs are appealing for numerous solutions. In the last decades, such protocols have been deployed at a large scale. A recent illustration of their popularity is the standardization of a PAKE in the new Wi-Fi Protected Access (WPA) standard, foreshadowing deployment on billions of devices in the following years [All21].

However, as the protocol's security now relies on the knowledge of a low entropy string, any leakage on a password-related value may compromise the authenticity and secrecy of the communication channel. This property makes them especially vulnerable to any side-channel attack, as we will demonstrate in our contributions.

As side channels continue to plague widespread implementations, the need for proper mitigations continues to grow. Researchers suggested various strategies to fix and undercut the spread of side channels. Namely, considering *software-based microarchitectural attacks*, contermeasures can be deployed at multiple levels: on the user-space layer, the system layer, or the hardware layer. Most generic solutions try to limit resource sharing or mask its access. Achieving resource isolation or access randomization at this level is challenging, but multiple solutions exist at either the system level or the hardware level. We refer the reader to the survey [LZJZ21], sections 5.1 and 5.2, for further insights about the state-of-the-art of such mechanisms. These mitigations would be a generic fix to these microarchitectural attacks. However, the proposed solutions are hard to deploy (*e.g.,* hardware changes) and may cause significant performance degradation on the entire system. A more affordable way is to aim for a patch at the application level, preventing the most common timing attacks targeting secret-dependent implementations. This can be achieved by enforcing a coding discipline called *secret independence*: all branches and memory accesses should only be dependent on public values. However, this property may be hard to guarantee, especially considering large and complex code bases. In fact, with the complexity of modern libraries, computer-aided verification is a promising way to eradicate secret dependence [BBB+21].

1.1. Contributions of this Thesis

In this thesis, we investigated multiple cryptographic implementations and noticed some recurrent types of vulnerabilities. Our research mainly focuses on analyzing widespread PAKE implementations, as they are particularly vulnerable to any form of side-channel attacks. To demonstrate the real estate of the attacks we discovered, we provided Proof of Concept (PoC) implementations leveraging software-based microarchitectural attacks. Considering these vulnerabilities and their scope, we investigated their origin and current solutions to prevent them. In our main contributions, we try to answer the following research questions:

I. Do PAKE protocols include specific protections against side-channels, considering their particular assumptions?

II. Why is today's cryptographic software not free of timing-attack vulnerabilities?

III. How can we provide guarantees against side-channel attacks at the software level?

The following contributions are divided into three categories. First, we analyzed two widely deployed PAKEs: Secure Remote Password protocol (SRP) [Wu09], a legacy protocol still deployed, and Dragonfly [Har08], a modern PAKE used in WPA3. In both cases, we found and exploited some leakage to recover the secret password, indicating a lack of awareness of the practicality of such attacks. Second, to understand the persistence of such leakages, we investigated the usage of computer-aided verification tools among cryptographic engineers. This revealed a low usage of automatic constant-time analysis tools in the real-world community, despite the prolific academic contributions. Gathering developers' feedback, we pinpointed the origin of this leaky deployment pipeline and established recommendations for both sides. Finally, because we could not find a scalable formal verification tool, we explored an alternative way to provide secret-independent implementation by generating formally verified C code. We developed a verified module to be integrated into a widespread Wi-Fi daemon, getting proper long-time mitigations for the vulnerabilities we discovered in Dragonfly's implementations.

1.1.1. Analyzing Implementations of Widespread PAKEs

The concept of PAKE is not new: it was initially described by Bellovin and Merritt in 1992 [BM92] and has received considerable attention since then. However, most existing literature focuses solely on the abstract design of such protocols, leaving their reference or deployed implementations aside. This is unfortunate, especially since passwords are particularly vulnerable to any secret leakage because of their low-entropy nature. Moreover, the RFC 8125 [Sch17] explicitly requires that any PAKE shall protect against timing and side-channel attacks. In this thesis, we aim to bridge this gap by scrutinizing the different implementations of two widely deployed PAKE: (i) the legacy SRP, and (ii) the Simultaneous Authentication of Equals (SAE) instantiation of Dragonfly used in WPA3. By studying both a legacy and a new PAKE, we aim to answer our research question **I**. We hope these contributions raise awareness concerning the need for (formally verified) constant-time algorithms and implementations that do not rely on incremental patches to guarantee security until the next leakage is found.

The Case of a Legacy PAKE: SRP

SRP is arguably among the most widely used PAKEs. It draws its popularity from its patent-free definition, quite straightforward design, and the availability of efficient open-source implementations. This contribution shows that numerous SRP implementations do not resist offline dictionary attacks. Indeed, we first identify some leakage vectors on the client-side of the OpenSSL SRP. Then, we exploit it through microarchitectural cache-based attacks. We simplify the execution of our exploit by automating the whole process, including spying on executed instructions, interpreting the noisy cache measurements, and recovering the password. Finally, we show how our attack concerns widely deployed solutions, such as Proton Mail and Apple HomeKit. For each project, we point out the vulnerable component and overall security implication. We notified the impacted projects and provided support in the patch integration. In total, seven patches have been developed and released following our work. Finally, we implement a PoC on different projects and make them open-source.

Reference: This work is the outcome of a joint work with Mohamed Sabt and Pierre-Alain Fouque. It has been published in the proceedings of *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security* [BFS21].

The Case of a Modern PAKE: Dragonfly

In 2018, the Wi-Fi Alliance announced WPA3-SAE to replace the Pre-Shared Key authentication defined by WPA2, that was subject to numerous attacks, notably KRACKs [VP17]. SAE is defined in the standard IEEE 802.11-2016 [IEE21], and it implements a slight variant of the Dragonfly RFC defined in [Har15]. Soon after its introduction, Vanhoef and Ronen presented Dragonblood, a set of vulnerabilities in WPA3 implementations, especially against its password-encoding method [VR20]. In response, the Wi-Fi Alliance published some implementation guidance to avoid attacks.

Dragonblood is Still Leaking. We identify several implementations in which some code is executed only when the password is correctly encoded. We show how an attacker can use cache attacks to leak some information on the password. Our attack builds upon previous work and drastically improves the performance of the original Dragonblood, extracting more information with fewer measurements. We apply our findings to the wireless daemon iNet Wireless Daemon (iwd)[1], from Intel, its related cryptographic library ell[2] and FreeRADIUS[3]. We have not only communicated the identified vulnerabilities to the maintainers of these two open-source projects but also helped them to patch the vulnerable code. In total, three patches have been developed and released. This contribution shows that despite following Wi-Fi Alliance guidances, Wi-Fi daemons may still be vulnerable to cache-based side-channel attacks.

Dragondoom: Leaking Wi-Fi Password Through Library Dependencies. After much effort, and four years after the standardization of WPA3 and amendment to the original protocol, one might reasonably presume that the attack vectors as presented in [VR20; BFS20a] be no longer relevant. Namely, eminent open-source WPA3 implementations (*e.g.,* hostap) are expected to be exempt from microarchitectural side-channel attacks, especially since they were analyzed both manually and by leveraging automatic tools to detect microarchitectural leaks, such as MicroWalk [WMES18a]. We challenge this belief and raise the question again of whether Dragonfly is secure in practice. We do not only point out another microarchitectural attack against Dragonfly. We bring to light an untapped source of leakage that still plagues WPA3 implementations. Indeed, we show that widely deployed daemons still suffer from side-channel vulnerabilities caused by their supported cryptographic libraries. Dragonfly implementations rely primarily on third-party libraries to perform their required cryptographic operations. These libraries do not always provide secret-independent implementations for their functions. Unfortunately, the security impacts of calling an external leaky function within Dragonfly were left unstudied. In this work, we analyze and track password-tainted values within the password conversion operation of Dragonfly, especially inside calls to external functions. Our analyses were rewarding: we noticed that most libraries we assessed are

[1] https://iwd.wiki.kernel.org/
[2] https://git.kernel.org/pub/scm/libs/ell/ell.git/
[3] https://freeradius.org/

vulnerable to side-channel attacks, leaking password-related values. Unlike previous works, we show that vulnerabilities do not only concern the legacy "hunting-and-pecking", but also the recently implemented hash-to-element based on Simplified Shallue-van de Woestijne-Ulas (SSWU), that is supposed to be secret-independent by design.

We demonstrate our vulnerabilities in multiple cryptographic libraries used by widespread SAE implementations: hostap, iwd and FreeRADIUS. We describe how the leakage can lead to an offline dictionary attack, which defeats the purpose of leveraging a PAKE protocol in WPA3. To this end, we use well-known microarchitectural techniques to monitor targeted instructions. We show the effectiveness of our attack by successfully recovering users' passwords in the last version of hostap using OpenSSL, which is the default setting for the Wi-Fi daemon on most Linux-based systems. We also provide valuable insight suggesting that other rising implementations, such as iwd/ell, are also affected by this vulnerability to a greater degree.

Reference: This work is the outcome of two collaborations.
The first is a joint work with Mohamed Sabt and Pierre-Alain Fouque. It has been published in the proceedings of *ACSAC '20: Annual Computer Security Applications Conference* [BFS20a].
The second is a joint work with Mohamed Sabt and Pierre-Alain Fouque. It has not been publicly presented yet and is described for the first time in this manuscript.

1.1.2. Investigating the Lack of Computer-Aided Verification

It is well-established that breaking cryptography is not merely a matter of theoretically breaking cryptographic algorithms. Implementation pitfalls are keys to guaranteeing that the running code does not leak any secret information. One class of vulnerabilities that interests us is one of the most widespread side-channel attacks: *timing attacks*. As we detailed before, in principle, these attacks are not that hard to mitigate. Indeed, we must ensure that memory access and control flow are independent of secrets. As new attacks were discovered, defensive research also evolved, and numerous technics exist to automatically verify implementations, supported by a broad range of tools [BBB+21]. These tools differ in their goals and offer various guarantees. Yet, they collectively demonstrate that automated analysis of secret-independent programs is feasible.

In this contribution, we set out to answer our research question **II**. To this end, we conduct a mixed-methods online survey with 44 developers of 27 popular cryptographic libraries. Through this survey, we track down the origin of the persistence of such side-channel attacks. Based on the survey, we identify shortcomings in existing analysis tools and issue recommendations for tool developers. Our findings show that poor tools resulted in secret-independence testing not being on the top of developers' to-do-list. The insight we got from the survey is that timing attacks might disappear or become scarce with more usable tools requiring little work overhead and giving easy-to-understand outputs.

Reference: This work is the outcome of a joint work with Jàn Jancar, Marcel Fourné, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. It has been published in the proceedings of *IEEE Symposium on Security and Privacy 2022* [JFB+22].

1.1.3. Formally Verified Implementation: an Alternative to Verification

In the previous contributions, we shed light on two serious issues impacting the security of modern implementations. First, side-channel attacks are indeed practical. Second, while rich, the current constant-time verification tool ecosystem lacks maturity in usability, hindering tools' deployment among cryptographic engineers.

Considering the residual vulnerabilities plaguing the cryptographic layer of WPA3 implementations, we aimed to verify its entire implementation to ensure that no other secret dependence was to be found. Overall, verifying a consequent subset of an existing library is a daunting task, and we could not find a scalable tool providing formal guarantees. However, even simpler tools, such as timecop [Nei; Lan10], revealed multiple leakages that could lead to password recovery attacks. Thus, between the lack of guarantees and the grueling task of investigating all possible secret dependence, we decided to tackle the problem from the other end, answering our research question **III**.

Instead of verifying an existing implementation, another approach consists in generating a formally verified implementation from well-defined specifications. Considering WPA3, the protocol itself can be divided into multiple layers, interacting with each other: the network layer, the protocol layer, and the cryptographic layer. We provide a formally verified C implementation of the cryptographic layer, guaranteeing secret independence, to serve as a plugin implementation in one of the most deployed Wi-Fi daemon: hostap. We built this plugin for hostap, where all the cryptographic calls within SAE redirect to the HACL* formally verified library [ZBPB17]. We provide the C implementation, achieving our security goals while keeping reasonable performance.

> **Reference:** This contribution is the second part of a publication based on a joint work with Mohamed Sabt, Pierre-Alain Fouque, Karthikeyan Bhargavan, and Natalia Kulatova. It expands our previous work on Dragonfly [BFS20a] but has not been published or publicly presented and is described for the first time in this manuscript.

1.2. Other Contributions

This section describes two contributions we excluded from the manuscript to maintain a coherent structure. Nonetheless, we discuss it here to draw a complete picture of all the contributions.

1.2.1. Revealing Vulnerabilities in SCP10 Standard

GlobalPlatform (GP) card specifications are the de facto standards for the industry of smart cards. Among other things, these specifications define the Secure Channel Protocol (SCP) family [Tec18]. As its name indicates, SCP channels prevent communication with the card from being observed or tampered with. SCP01, SCP02 and SCP03 are symmetric keys ciphering mechanisms using DES (used in SCP01), Triple DES with two keys (in SCP02) and AES (in SCP03). GP also defines two SCP channels relying on asymmetric-key cryptography: SCP10 and SCP11. Despite their importance, the SCP protocols have been overlooked in the literature for a long time. To the best of our knowledge, Sabt *et al.* provided the first cryptanalysis for both SCP02 and SCP03 [ST16]. They proved that SCP03 provides stronger security guarantees by preventing Algorithm Substitution Attack (ASA) [BJK15]. They

also showed a theoretical attack against SCP02 due to misuse of Initialization Vector (IV). In [AF18], Avoine *et al.* showed a more practical attack against SCP02, in which they exploit CBC padding oracles to recover some secret data.

We study the specifications of SCP10 that is becoming more relevant to deploy given the deprecation of SCP02 and an ecosystem increasingly open. Our analysis identifies various design flaws in SCP10 that allow attackers to mount practical attacks. Indeed, we note that SCP10 encrypts data without any random padding. Then, we note that it reuses the same RSA key pair for encryption and signature. We leverage the fact that PKCS#1 v1.5 [MKJR16] is still used for signature to decrypt sensitive messages by applying both the Coppersmith [Cop96; Cop97] and Bleichenbacher [Ble98; BFK+12] attacks in different settings. Using real smart cards, we evaluate the complexity of our attacks. Interestingly, a padding oracle attack originating from a timing difference in processing is likely to affect many implementations given the lack of specification, uncommon padding, and constraint environment. Finally, we propose a secure implementation of SCP10 specifications that complies with the latest National Institute of Standards and Technology (NIST) recommendation in terms of Key Agreement [BCR+19]. GlobalPlatform was responsive regarding our analysis. As a result, we lead a working group with their partners to mitigate the identified weaknesses. This process results in an amendment to SCP10 providing better security [Tec20].

It would be interesting to extend our work to SCP11, which relies on elliptic curve cryptography. Although SCP11 is still defined only as an amendment, existing works within GP reflect its foreseeable future adoption for embedded SIM cards. Unfortunately, to this date, no known analysis has examined the security of SCP11.

Reference: This work is the outcome of a joint work with Mohamed Sabt and Pierre-Alain Fouque. It has been published in the proceedings of *IACR Transactions on Cryptographic Hardware and Embedded Systems 2020* [BFS20b].

1.2.2. Chosen Group Attack Against ProtonMail Authentification

ProtonMail is a well-known mail solution that built its reputation around its investment in security. Namely, it distinguishes itself from other solutions by providing automatic client-side email encryption (protecting users from a malicious server) and by open-sourcing its code. Of particular interest to us, user authentication relies on SRP to get the authentication token, enabling access to account management and possibly (depending on the version) encrypted emails.

We took an interest in a particular application: their python client[1].

Abusing peculiar behaviors of two module dependencies (pySRP and python-gnupg), we were able to bypass the signature verification on group elements for the SRP session and provide custom parameters. Using wisely chosen parameters, attackers can solve the Discrete Logarithm Problem (DLP), recover user ephemeral key share, and perform dictionary partitioning attacks on SRP.

Given the implementation details, the threat model considers either a malicious server or attackers able to bypass certificate pinning (which is only available for ProtonVPN at the time of testing) and having their ssh public key in the victim's keyring.

More details on the attack are provided in Appendix A.

[1] https://github.com/ProtonMail/proton-python-client

1.3. Artifacts and Patches

Along with our various contributions, we attached particular importance to the reproducibility of our results. In that sense, we provide artifacts in separate repositories, primarily as PoC. In addition, we provided patches or feedback on mitigation to projects affected by our attacks. Below, we list the practical contribution that stems from our work on different topics.

SCP10. In subsection 1.2.1, we describe two attacks on the session establishment of SCP10 on a smartcard. We implemented both and publicly disclosed the applets used to mount the padding oracle attack and the key recovery attack. Along the vulnerable applet, we implemented the suggested mitigations to get a practical sense of the overhead they incur. All applets are available at `https://github.com/ddealmei/SCP10-attack`.

The issues we disclosed lead to an amendment to the standard.

Dragonfly. For both contributions revealing attacks on Dragonfly (subsection 1.1.1), we provide all data and attack scripts with a Dockerfile. The setup enables easy reproduction of the entire attack pipeline, from leakage aggregation to password recovery. We also provide our dataset of leaked information.

- The first attack, demonstrated on iwd/ell, is available at `https://gitlab.inria.fr/ddealmei/poc-iwd-acsac2020`.
- The second attack, demonstrated on hostap and OpenSSL, is available at `https://gitlab.inria.fr/ddealmei/artifact_dragondoom`.
- A formally verified cryptographic provider, with operations necessary to implement Dragonfly, is available as a plugin to hostap. The repository contains a Dokerfile to avoid dependency issues and ease building. It embeds testing data (differential fuzzing with OpenSSL) and benchmarking scripts. The artifact is available at `https://gitlab.inria.fr/ddealmei/artifact_dragonstar`.

In addition, multiple patches and security advisory have been issued by hostap (see security advisory), iwd/ell (see discussion threads [1], [2], [3] and [4]) and FreeRADIUS (see changelog and commits [1] and [2]) to fix the vulnerabilities.

SRP. For the contribution on SRP introduced in subsection 1.1.1, the PoC's artifact exploiting a side-channel in OpenSSL's implementation of SRP is available at `https://gitlab.inria.fr/ddealmei/poc-openssl-srp`.
The attack is implemented on OpenSSL, pySRP and Apple HomeKit to demonstrate its scope. Following the disclosure, we contacted all affected projects we could find and supported them through their mitigation process by submitting patches (on OpenSSL, Apple HomeKit, pySRP, ProtonMail's python client, node-bignum, ruby/openssl and Erlang OTP).

1.4. Outline

This thesis is divided into three parts, excluding this introduction. The first part introduces some background on key notions that will be central to most contributions :

- Chapter 2 gives an overview of modern microarchitectures. It provides a more detailed explanation of CPU caches, often abused to exploit microarchitectural side-channel attacks. Finally, it introduces the main *cache-timing attacks*.
- Chapter 3 presents the evolution of PAKE protocols, and the standard assumptions related to their threat model. Then, it describes the two protocols that are central to this thesis: SRP and Dragonfly.
- Chapter 4 describes why any leakage on PAKEs may lead to password recovery and present a generic dictionary partitioning attack. It also defines a metric to evaluate the efficiency of a side channel based on the leakage, that we will use to compare to previous work. The chapter ends with a summary of recent side-channel attacks on PAKEs.
- Chapter 5 concludes the preliminary chapter with an insight on related topics that will be omitted and a presentation of the default threat model that we will consider across the thesis.

Next, the core of the manuscript comes down to Part II, detailing all our contributions:

- Chapter 6 (resp. chapter 7) describes multiple vulnerabilities we found in widespread SRP (resp. Dragonfly) implementations. For each vulnerability, we give a theoretical evaluation of the leakage and a detailed explanation of the attack. We also provide an impact analysis on affected projects and mitigations strategies.
- Chapter 8 presents our study of the constant-time verification tool adoption among cryptographic developers. This chapter differs from previous ones as it contains fewer technical details and focuses on the user study perspective to provide valuable feedback to both tool developers and library maintainers.
- Chapter 9 focuses on an alternative path to secure implementation: generating formally verified implementations. It describes the fundamental of the formally verified HACL* library and how we leverage it to provide a sustainable, secure implementation of the cryptographic operations needed to implement Dragonstar. We build this to be integrated into hostap as a plugin cryptographic provider.

Finally, the last part (Part III) concludes and gives some perspectives on future works.

How to read. Each chapter starts with a summary of its content and a *Takeaway box* highlighting the key aspects and the outline. In Part II, the contribution part, each chapter will point to the related background at the start. However, all chapters are self-contained as a short reminder of key concepts will be provided in *Reminder boxes*, as displayed below.

> **Reminder:** This is a reminder box.

At the start of technical sections, (*e.g.,* containing detailed computations), small *Takeaway boxes* sums up the content of the section. Hence, the reader can skip the section without missing the point, or read the section with an idea of where they are going.

> **Takeaway:** This is a takeaway box.

Most contributions are heavily related to the background on cache timing attacks (mainly Flush+Reload) introduced in section 2.3 and the generic partitioning attack described in chapter 4. The knowledge of PAKEs provided in chapter 3 may help to grasp the scope of the contributions described in chapter 6, chapter 7 and chapter 9. Chapter 8 only requires a shallow understanding of constant-time analysis tools, and reading the background part is not required to understand our contribution. Similarly, the necessary background to apprehend the formal aspect of chapter 9 is provided in the same chapter.

We try to keep the writing gender-neutral by opting for the plural pronoun "*they*". Hence, readers may encounter phrasing such as "The attacker leaks information when *they* make a measurement.".

Part I.

Preliminaries

Modern CPU Microarchitectures and Related Attacks 2

Modern processors are highly parallelized units capable of processing billions of operations per second. This performance is achieved by the growing technological capabilities of constructors and by combining numerous optimizations. Here, we present some elements involved in executing classical instructions on a modern Central Processing Unit (CPU), as well as various microarchitectural optimizations that cause side-channel leakages. Then, we introduce the most notorious microarchitectural attacks based on the contention on a specific element of the microarchitecture: CPU caches. The microarchitectural background is heavily oriented toward presenting Flush+Reload-like attacks and thus omits many details and components. We only consider Intel processors for clarity and to keep the background relevant to the following chapters.

Contents

2.1. Optimizing the Execution **18**
2.2. Memory Layout and Cache Hierarchy **19**
2.2.1. Shared Memory and Page Deduplication 20
2.2.2. CPU Caches 20
2.3. Cache-Timing Attacks **23**
2.3.1. Prime+Probe 24
2.3.2. Flush+Reload 25

2.1. Optimizing the Execution

This section provides an overview of a generic Intel CPU microarchitecture. We do not aim at dressing a complete picture of the execution pipeline but giving the reader an insight into how we get from a binary program to the actual execution on the CPU.

We start with the initial assumption that the sequence of instructions defining a program is stored in Random Access Memory (RAM). When executing the program, both instructions and potential operands are sent from the RAM to the CPU to be processed. Before being fed to the execution unit, they are processed by a front end which decodes each instruction into micro-operations (μops), to be processed by hardware circuits.

Besides the high clock frequency, processing instructions and data cleverly can spare cycles on the execution. Instead of processing everything linearly, modern processors rely on pipelining to break down the complete execution of instructions into several processing units. This yields a complex yet efficient execution pipeline. The different stages of the pipeline usually process in parallel but depend on each other. Therefore, the overall execution is at most as fast as the slowest component of the pipeline. Several elements, such as memory delays, data or control hazards, and contention on shared resources, can cause slowdown and stalling. Alongside, multiple optimizations are in place to minimize the performance cost. Below, we describe some of the mechanisms in place to avoid stalling.

Simultaneous Multithreading (SMT). Modern computers usually benefit from multiple physical cores, each benefiting from its execution pipeline built upon different hardware functional units. To get the most out of each functional unit, *Simultaneous Multithreading (SMT)* [TEL95] allows multiple threads to execute in parallel on the same hardware component. Hence each physical core can be divided into multiple logical cores, sharing the same hardware resources. This sharing is entirely transparent from the software perspective and makes it possible for instructions of different threads to be processed by any pipeline unit.

Out-of-Order Execution. Data dependencies may occur if a stage is waiting for the outcome of a previous pipeline stage (*e.g.,* waiting for an operand to be fetched from memory). In this case, the functional unit would idle, and the processing of all future instructions would be delayed. Instead of stalling, a stage may process instructions used later in the execution if their operands are available. Thus, instructions are issued based on their order in their program, then enter a queue where they are processed based on the availability of their operand(s). This *out-of-order* execution allows the CPU to keep working instead of waiting, and to get ahead of the linear execution. For the instructions to be processed out-of-order, they must be independent of previous results, which may not be processed yet.

Branch Prediction. A program can take multiple paths based on the value of a variable, creating a branch in its control flow. A naive processor would need to wait for the value to be resolved to know which branch to take. With the complex pipelining of modern CPUs, this control-flow dependency would considerably slow down the overall execution. A branch predictor is a functional unit that attempts to guess which branch is the most likely to be taken and continue the execution following that path instead of stalling. This approach is risky since mispredictions significantly hinder execution performance. In such cases, changes must be reverted so that the incorrect instructions have no effect, and the correct instructions

must be resolved and fetched. The longer the pipeline is, the more cycles it takes to revert the changes. Thus, a complex pipeline justifies the need for advanced branch prediction strategies.

CPU caches. Memory delays can occur when the input of a stage is not yet available. Fetching data and instructions from memory can take a couple of cycles up to hundreds of cycles, depending on where it is located. Fetching each instruction independently would cause a considerable performance bottleneck. To avoid this cost, the CPU benefits from *caches*. Caches are a particular hardware feature that offsets the discrepancy between fast processing and slow memory access. They act as small but fast access buffers located directly on the CPU. Data is fetched from higher memory levels into caches 64 bytes at a time. When executing a program, CPU cores will look into its lower cache for instructions and data. If the cache line is not available (cache *miss*), it will be fetched from higher memory levels (*e.g.,* second-level cache (L2), Last Level Cache (LLC) or RAM). Then, if the CPU re-access it later, it should find it in the cache (cache *hit*), avoiding the overhead of accessing higher-level memory and reducing the latency. Since it is an essential part of the attacks we implemented, we give further insight into the cache structure and overall memory management in section 2.2.

Prefetchers. To avoid the systematic cache miss on first access, causing additional delays in the execution, *prefetchers* were built upon caches. They aim at predicting data accesses and fetching the cache lines ahead of time, thereby reducing memory delays. We can find multiple prefetching mechanisms based on either software or hardware. While the former is introduced at compilation time, the latter is provided by the CPU itself. Here, we focus only on the latter, particularly on recent Intel cores.

Intel documentation [Int21] classifies data reference patterns in three categories: (i) *temporal* if data will be used again soon; (ii) *spatial* if data will be used in adjacent locations (*e.g.,* on near cache lines); (iii) *non-temporal* if data is referenced once and not reused in the immediate future. Multiple prefetching techniques are defined to handle a wide variety of applications and get the best performance.

Stream prefetching [Jou90; PK94] detects consecutive access to memory and fetches multiple lines ahead into L2, within a memory page boundary. *Adjacent-line* prefetching [JMH97] fetches lines based on the spatial locality of other cache lines. It will fetch cache line pairs to L2. *Next-line* [Smi82] technique will simply fetch the next cache lines to data cache (L1d), based on spatial locality. Finally, *stride* prefetching [FPJ92] consists in extracting a pattern from accessed addresses, based on an access table. It will fetch the predicted cache lines in L1d.

Intel recently documented [Int21] the use of these techniques, with four different hardware prefetchers: the *streamer*, *Spatial prefetcher*, *Data Cache Unit prefetcher*, and *Instruction pointer-based prefetcher* respectively.

2.2. Memory Layout and Cache Hierarchy

Optimized memory management, whether at the system or hardware level, is crucial to efficient execution. Interestingly, optimizing its management involves shared resources and potential contention elements that an attacker may exploit. First, we describe how the memory of processes (instructions, data, or shared libraries) is handled by the Operating System (OS),

creating shared memory for read-only data. Then, we give further details on CPU caches' internal functioning.

2.2.1. Shared Memory and Page Deduplication

Each running process has access to a virtual memory range, representing an abstraction of the storage resources available on a computer. The OS manages the mapping between physical and virtual memory. Modern systems implement a paging mechanism to efficiently load physical memory from the disk into RAM by fixed-sized blocks of at least 4KB called *pages*.

In a multi-processes environment, multiple users or processes may use the same code or read-only data in different processes (*e.g.,* a shared library), causing memory redundancy. To avoid saturating the RAM, modern OSs identify identical pages based on their physical location. Then, they proceed to a *content-based page deduplication*, mapping identical addresses to the same virtual page. Hence, multiple processes may share the same memory space if they access the same library or fixed data. To avoid malicious processes modifying shared data, only `read` operations are allowed and a `write` triggers a CPU interrupt. In such cases, the OS would interrupt the execution and copy the shared page's contents in a newly allocated page in the process's address space before letting the process continue its `write`. This technique, called *copy-on-write*, is implemented on all modern OS to limit memory allocation.

A more aggressive approach, called *content-aware page deduplication*, consists of scanning active memory to look for and merge pages with identical content. This is primarily used in virtualized environments where multiple guests co-exist on the same host, thereby sharing the same physical memory.

2.2.2. CPU Caches

Cache Hierarchy. On modern hardware, caches are broken down into multiple levels, built with an access hierarchy, going from the lower first-level cache (L1) to the LLC, shared across all cores on modern CPUs. On personal computers, the LLC is usually L3. First, the L1 is the closest to the execution core. It is the only cache where data and instructions are separated (into L1i and L1d for instructions and data, respectively). Along with the L2, they are located on the same physical core and not shared with other cores. On the other hand, the LLC is divided into multiple *slices*, each dedicated to a core and linked by the interconnect ([Int21], Section 2.4.5.3, defines the interconnect as a bidirectional ring between cores). Finally, the LLC is connected to the RAM, as shown in Figure 2.1.

Cache Structure. Each cache typically stores 64-byte blocks of adjacent data in the form of *cache lines* and is organized in multiple *cache sets*, each composed of several *ways*. Breaking down the structure of caches in such a way makes it easier and more efficient for the execution unit to know whether a specific piece of information is cached.

It is important to distinguish between a memory address, informing the CPU on the location of the data to access in the cache, and the cache line, containing the actual data. Figure 2.2 represents their respective structure.

A memory line (Figure 2.2a) usually contains three fields: (i) a *tag* acting as an identifier for the corresponding cache line; (ii) the *index* of the cache set containing the data, computed from the physical address; (iii) the *offset* of the byte in the cache line.

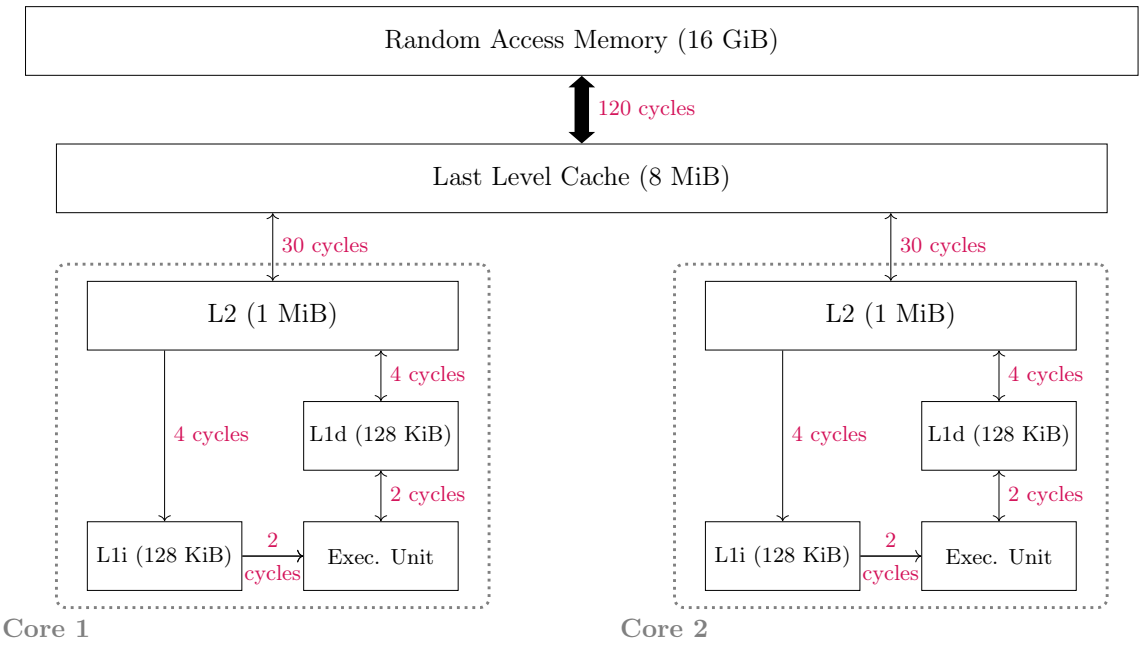


Figure 2.1. – High-level overview of the interaction between the memory subsystem and higher levels of memory. Each arrow represents a data flow, and the red number represents a usual latency. Note that these latencies, and cache sizes, were observed on an Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, but may vary for other processors.

A cache line (Figure 2.2b) does not only contains the data. It is prefixed by a *tag* based on its memory address, which makes it possible to look for the tag instead of comparing the data block. Furthermore, the data block is suffixed with a *flag* describing its coherence-state (described later in this section).

When the CPU needs to access a memory address, it starts by looking for it in the cache. It first maps the memory address to the corresponding set by consulting the *index* field (Figure 2.2a). The line may then be anywhere within that set. The capacity of a set balances the cost and performance of the search. In a *direct-mapped* cache, each set contains a single cache line. On the other hand, a *fully associative* cache has a single set, as big as then entire cache. Often, sets can contain multiple lines to balance the cost and performance of both approaches. A set is called *n-ways* if it can contains n cache lines. In the latter scenario, the CPU looks for the line matching the tag within that set. If a match is found, and the flag valid, it is a *cache hit*, and the *offset* field informs the CPU about which byte to look for in the cache line. If there is no match, or the flags indicate an *invalid* line, it is a *cache miss*, and the line must be fetched from higher memory levels (L2, LLC or RAM).

The size of caches is limited, so a set may be full, with two lines competing for the same place in the cache, resulting in a collision. This justifies the need for a *replacement policy* defining the condition for a cache line to be evicted and replaced by another. An intuitive and standard policy, Least Recently Used (LRU), replaces the less used cache line, as it is not expected to be re-used soon. Other architectures, namely on ARM processors, pseudo-randomly choose the line to evict, reducing the choice complexity as it avoids keeping track of cache line usage. However, vendors do not document the exact procedure, and shared knowledge comes from



(a) Memory address structure in a set-associative cache.



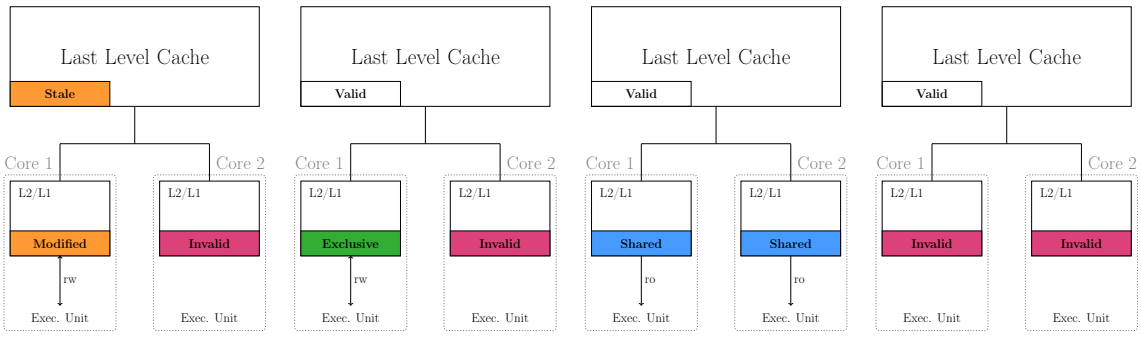
(b) Cache line structure.

Figure 2.2. – Representation of memory address and the corresponding cache line. The tag is the same in both figures. The sizes of the fields are not to scale.

the reverse engineering efforts of researchers [HWH13].

Inclusiveness. Most of Intel CPUs use *inclusive* caches. Thus, any cache lines stored in lower-level caches are also stored in the LLC. A side effect of that property is that coherency must be enforced when flushing a cache line. Hence, if a line is flushed from a cache level, it has to be flushed from all cache levels. Other architectures may rely on different strategies. We observe an opposite behavior in *exclusive* caches: data cannot be in two cache levels simultaneously. Finally, with *non-inclusive* caches, data may be present in multiple caches without strict coherence enforcement.

Cache Coherency. In a multi-core setting, because of data sharing, the CPU needs to keep track of the state of cache lines in the private caches (*i.e.,* in L2/L1). Hence, a coherence protocol is needed to preserve data consistency and ensure any change in a shared operand is propagated to caches containing them. The value of its flag describes the state of a cache line (Figure 2.2b)).



(a) Modified. **(b)** Exclusive. **(c)** Shared. **(d)** Invalid.

Figure 2.3. – Different cache line states based on the MESI coherence protocol.

Most modern processors rely on the MESI coherence protocol (or a variant), described hereafter (Figure 2.3). Following this scheme, a cache line can take four states:

1. ***Modified***: the core has changed the value of the line in the private cache. Its value in higher memory levels must be updated before the *dirty* line is evicted. The state of this line in other private caches is changed to *invalid* (Figure 2.3a).

2. ***E**xclusive*: the cache line value is correct and only available in this private cache (Figure 2.3b). A `write` would cause the line to switch to the *modified* state without the need to preserve coherence with other private sets.

3. ***S**hared*: a clean copy of the line is available in multiple caches (Figure 2.3c). A `write` would switch its state to a *modified* and trigger the coherency process to make it *invalid* in other caches.

4. ***I**nvalid*: the value of the line is not valid, and any access would result in a miss. Accessing this line requires looking for it in higher memory levels, starting with the LLC. If the line in the LLC is valid (Figure 2.3b or Figure 2.3d), the cache line is updated accordingly. Otherwise (Figure 2.3a), the coherency protocol updates the LLC with its most recent value from another L1 cache, and passes on the values to the L1d.

2.3. Cache-Timing Attacks

As CPU resources are shared among processes, an attacker may abuse them to recover part of its internal state, thereby inferring information on the data or instructions being processed. A recent example of such attacks is Hertzbleed [WPH+22], allowing the attacker to deduce information on the data being processed from the overall execution time because of the dynamic frequency scaling of modern x86 processors. Many other microarchitectural attacks leverage contention on a specific shared component, such as the bus interconnect [PLF21], the TLBs (Translation Lookaside Buffers) [GRBG18], ports of the execution units [ABH+19; RMBO22], or, more relevant to our contributions, caches. In the following, we focus on the last resource and give an overview of the main attacks leveraging cache contention. We refer to them as cache-timing attacks, or simply *cache attacks*.

These attacks have often targeted cryptographic implementation since side-channel leakage usually represents the most cost-effective way to break the security properties of such protocols. However, they have also been applied in a more generic way to build covert channels between processes [MWS+17] (breaking the isolation) or to fingerprint websites and users [SAO+21; LMD+22].

To achieve their goal, these attacks exploit a variety of software tricks to control the state of the cache. The attacker can then abuse their knowledge of particular cache features to infer information through the timing difference in executing specific instructions. Namely, interacting with a memory line can take a different amount of time depending on whether it is present in the cache or not. The access is fast if the memory line is already cached (cache *hit*). Otherwise, if the cache line is not available (cache *miss*), it will be fetched from higher memory levels (*e.g.,* L2, LLC, RAM), and the latency increases for each memory level.

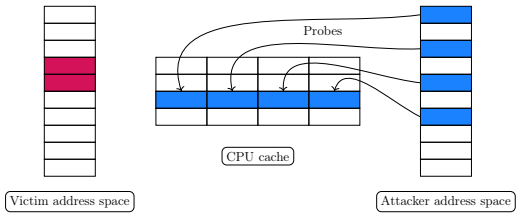
Cache-timing attacks can be exploited through software-only interaction with the victim's computer and do not require physical access to the machine. Excluding particular scenarios (like attacking secure enclaves), a typical threat model for such attacks assumes the attacker can execute some unprivileged code on the targeted processor. Co-location assumes the attacker run code on the same physical core (cross-thread attacks) or on a different core in the same processor (cross-core attacks).

Below, we present a non-exhaustive list of the most notable cache-timing attacks of the last decades. Each technique may achieve different precision (spatial and temporal) and be more suited depending on the targeted microarchitecture. Hence, the choice of method is crucial to

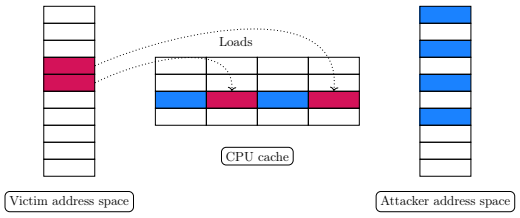
the success of an attack. We will consider two protagonists in our attack scenarios: a victim $\mathcal{V}$ and a spyware $\mathcal{S}$.

2.3.1. Prime+Probe

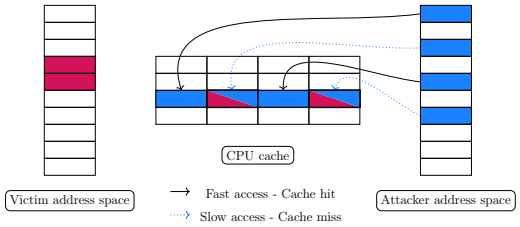
Introduced in 2005 [OST06], Prime+Probe reveals memory access patterns of a process running on the same core by leaking the cache set accesses. The attack was later extended to target the LLC, making cross-core attacks possible on inclusive caches [LYG+15]. This technique shines by its low requirements, making it particularly relevant when the attacker has limited control over the victim's machine. Of particular interest, the attacker needs neither shared memory nor direct control over the cache with a flush instruction (unavailable on ARM processors). These weak assumptions also make it an excellent candidate for browser-based attacks, with an attacker controlling Javascript code on web pages [OKSK15].



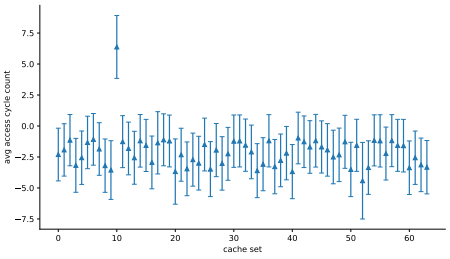
(a) The attacker primes the cache



(b) The victim uses the cache when running



(c) The attacker probes the cache



(d) Example of the output with access to cache set 10 (generated with CacheSC).

Figure 2.4. – Prime+Probe attack illustrated. The attacker is monitoring the victim's access patterns to a specific cache set. **(a)** The attacker primes (*i.e.,* fills) the targeted cache set. **(b)** the victim evicts some cache line during its execution. **(c)** The attacker probes their data to determine if the victim has accessed a specific set. If the victim used this set, they have evicted some of the attacker's data, causing a longer access time during the probing step.

The attack workflow, illustrated in Figure 2.4, only assumes that the attacker can execute code on the same processor as the victim. First, $\mathcal{S}$ fills some well-chosen cache sets with their data. It then idles while $\mathcal{V}$ continues to execute. Finally, $\mathcal{S}$ re-accesses the same cache sets. If $\mathcal{V}$ accessed a set during its execution, it would have evicted some of the attacker's data, making the probing longer.

The attack involves several challenges that must be overcome to make it efficient. First, the attacker must identify the relevant cache set corresponding to critical data. To get a reliable outcome, they monitor each cache set and look for consistent patterns corresponding to the

victim's activity. Second, the attacker has to build an efficient *eviction set*. That is a group of memory lines, in their virtual address space, that would map to the relevant set(s) to monitor. While the lower level L1 is virtually indexed, making the mapping trivial, attacking the physically-indexed LLC is more challenging. In the latter scenario, the attacker must partially recover the physical address of memory lines while the mapping is not publicly known. They can overcome this issue statically or dynamically. The former implies reverse-engineering the mapping function, which may be processor specific [HWH13]. The latter consists of looking for conflicting memory lines and has been improved over time [LYG+15; VKM19; SL19].

However, the low requirements of PRIME+PROBE come at a price: the spatial and temporal resolutions of the attack are hindered by the lack of an accurate measurement method. Given its low resolution, this technique is more adapted to side-channel relying on memory access than instruction-driven leakages. As a result, it has been mainly used to exploit leaks in the data flow (*e.g.,* AES T-table implementations [NS06; OST06] or RSA with pre-computed tables [IGI+16]).

Temporal limitations. Building an efficient (*i.e.,* small) eviction set may be time-consuming and need to be built at runtime, but it can be done once for each probe. However, the attacker must probe the entire set whenever they make a measurement. Depending on the size of the eviction set, and the targeted cache level, this process may take hundreds to thousands of cycles. Meanwhile, they are unable to track the victim's behavior.

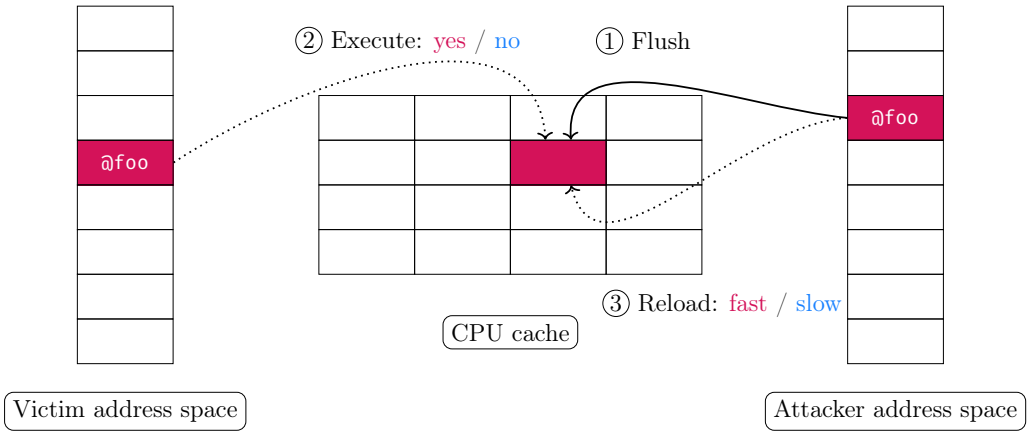
Spatial limitations. Using such a technique, the attacker constructs access patterns to a particular *set* of a cache but cannot distinguish which of the multitude of memory locations that map to the cache set has been accessed.

Variant: PRIME+SCOPE [PTV21]. Akin to PRIME+PROBE, the threat model of this attack is minimalist: the attacker must be able to execute code on the same CPU as the victim. In previous attacks, an attacker introduced the observer effect while making measurements: their probing affected the state of the cache, decreasing the accuracy of the results. This new approach overcomes this effect and achieves a better time resolution while maintaining the wide portability of PRIME+PROBE. The main improvement consists of replacing the probing of a complete set in the LLC with a single cache line access in L1, corresponding to the eviction candidate. This permits fast and frequent measurements in the scope step. Nevertheless, whenever an event is detected, the cache state still needs to be reset by the attacker, which can take thousands of cycles (compared to about 70 cycles for an eventless measurement). Hence, this technique is still not suited for high-frequency events. As an additional requirement, PRIME+SCOPE needs $\mathcal{S}$ not to be executed on the same physical core as $\mathcal{V}$, and a predictable replacement policy to guess an eviction candidate reliably.

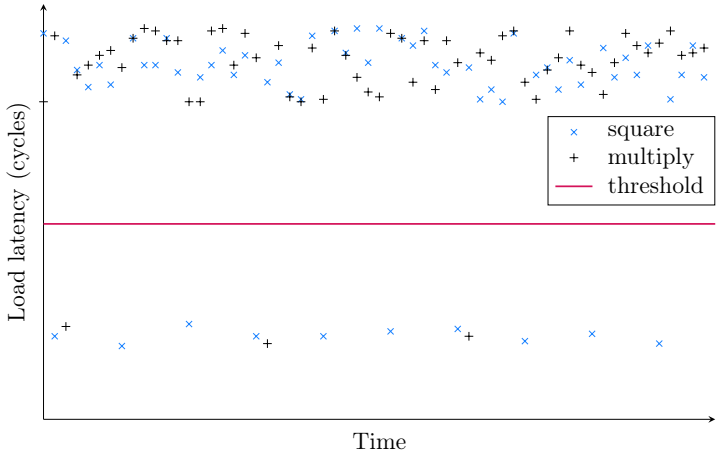
2.3.2. FLUSH+RELOAD

Introduced in 2011 [GBK11] and generalized in 2014 [YF14], FLUSH+RELOAD is a cross-core attack leveraging the time gap between a cache hit and a miss on the LLC to get information on the cache state and infer a particular process access pattern. This attack, referred to as *access-driven*, is particularly effective in guessing the program execution flow.

The idea, illustrated in Figure 2.5, is to exploit memory sharing between two processes - a victim $\mathcal{V}$ and a spyware $\mathcal{S}$. The goal of $\mathcal{S}$ is to probe a specific memory line from this shared



(a) FLUSH+RELOAD's 3-steps workflow. The attacker is monitoring the victim's access to *foo*. ① the attacker flushes a memory address, shared with the victim, out of the cache; ② while the attacker waits, the victim may load the probe; ③ the attacker reloads the probe and observes the reloading latency: a fast reload means the victim has accessed the probe.



(b) Output of a FLUSH+RELOAD attack on a naive square-and-multiply implementation, revealing bits value. Each cross under the threshold represents a cache hit (fast reload). Observing squaring distinguishes each bit processing, if a multiplication is observed, the bit is a `1` (1001001000_2).

Figure 2.5. – FLUSH+RELOAD attack illustrated.

memory. The attack is carried out in three steps. First, $\mathcal{S}$ flushes the probe out of the LLC with the appropriate instruction (`clflush` or `clflush_opt`). Since the LLC is inclusive, this operation ensures that the probe is no longer in any cache. $\mathcal{S}$ then idles for a pre-determined period, during which it waits for $\mathcal{V}$ to execute - or not - the probe. Finally, $\mathcal{S}$ reloads the probe and measures the reload time. If $\mathcal{V}$ has executed the probe during the idle period, the reload results in a short reload time, hence a hit. Otherwise, the access will be significantly longer. Repeating these steps allows the attacker to identify when the victim has used the probe.

Despite being very powerful, Flush+Reload suffers from some limitations.

Temporal limitations. The mere idea behind the attack is to perform a few assembly instructions to flush, reload and time the operations. Naively looping over these operations would introduce blind spots on concurrent reload, where the attacker would miss many events. To prevent this, measurements are usually performed periodically by defining an appropriate idle period between flushing and reloading. Doing so, on the one hand, we increase the temporal resolution by the additional idle. On the other hand, the attacker would not distinguish multiple accesses to the probe if they occur within the same time slot. Slowing down the execution with a Performance Degradation Attack (PDA), by repeatedly flushing out a well-chosen cache line, allows them to benefit from a longer idle without missing valuable information [ABF+16]. This technique was then extended, going from DEGRADE to HYPERDEGRADE, by pinning the degrading thread to the same core as the victim. By doing so, the thread benefits from the spatial proximity of the shared L1 cache, increasing the flushing rate of the PDA [AB22]. Combining Flush+Reload with a PDA has become the default use of the attack, and has been employed numerous time within the last years [ABF+16; GBY16; GB17; BBG+17; PBY17; AGTB19; ANT+20; CKP+20; GHT+20; BFS20a; BFS21].

Spatial limitations. Flush+Reload is designed to detect when some instructions are loaded into the CPU cache. This allows monitoring of the memory activity of the victim process at a cache line granularity. However, widespread CPU optimizations, such as prefetching, need to be considered. Namely, the prefetcher will, among other things, fetch adjacent cache lines to avoid unnecessary lookups. Hence, the original cache line granularity might be impeded by making the relevant instructions loaded into the cache before the CPU would execute them. Speculative execution might also cause trouble to Flush+Reload since instructions directly next to a condition might be loaded into the execution pipeline (hence the CPU caches) regardless of the condition output. However, the attacker usually circumvents this limitation by targeting instructions further down the execution to be sure the cache hits do not correspond to a missed speculative execution. These limitations were exposed in the seminal work [YF14], and were considered in later works [YB14; ABF+16].

Variant: Flush+Flush [GMWM16]. This technique introduces a stealthier and faster alternative to the vanilla Flush+Reload. Instead of observing the latency of reloading the probe, this attack relies on the latency difference of the `clflush` instruction. Its execution time depends on the presence of the memory line on the cache (slower when a memory line is cached) and the coherency state of the line. Since the average latency of flushing is an order of magnitude lower than a cache miss, this change makes the attack faster. Additionally, the attacker avoids most changes to the cache state, bypassing detection techniques relying on

cache hit/cache miss counters. The main drawback of this approach is that the threshold to differentiate from a `clflush` with and without data available is much lower than for Flush+Reload and may be difficult to distinguish in practice. Recently, Didier and Maurice [DM21] provided a better calibration technique to increase the accuracy and avoid noisy results.

Variant: Prefetch+Reload [GZZY22]. This attack improves spatial and temporal resolution with a similar threat model as Flush+Reload. It exploits the cache coherency state of read-only data to learn whether some data is cached. Interestingly, the authors noticed that the instruction `PREFETCHW` sets the coherence state of a single cache line to *Modified*, triggering the coherence protocol to invalidate copies of that line in other private caches.

When controlling two threads - $\mathcal{S}_1$ and $\mathcal{S}_2$ - on different cores, an attacker can then detect the victim's access to the line by differencing the latency of a *private* cache hit (hit from L1 or L2), and a LLC hit. The first thread, $\mathcal{S}_1$ calls `PREFETCHW` on the probe, invalidating its state in all other private caches, including for the victim and $\mathcal{S}_2$. If the victim tries to access this cache line, the coherence protocol will update its state to *shared* in all caches, and $\mathcal{S}_2$ may access the clean line in its private cache. Otherwise, $\mathcal{S}_2$ needs to update the line from the LLC, resulting in a slightly longer latency.

Using this approach, an attacker track the victim's access by distinguishing private cache hits (if the victim accessed it) from LLC hits (if it is still *invalid*). In addition, the `PREFETCHW` instruction is applied to a single cache line, avoiding triggering noisy CPU optimizations.

Password Authenticated Key Exchange 3

Passwords are by far the most popular method of end-user authentication on the web. However, password breaches, due for instance to sophisticated phishing attacks, seriously multiply the risk of authentication compromise. Aware of such issues, a growing number of service providers couple passwords with another authentication factor: this is known as two-factor authentication (2FA). However, a longitudinal analysis in [Bur18] highlights that the adoption rate of 2FA has been mostly stagnant over the last years. This is partly because 2FA requires unwilling users to modify their way of interacting with the web. An alternative approach is to keep relying solely on passwords but to change how they are verified to protect against phishing attacks, server breaches, and dictionary attacks. The advantage of such an approach is that it does not change users' experience. However, this does not come without cost; the burden is now shifted to service providers that should support new cryptographic protocols. These protocols are called Password-Authenticated Key Exchange protocols (PAKE).

In this chapter, we present some common properties of PAKEs, and define a taxonomy relevant to their vulnerability to side-channel attacks. Then, we describe the two key protocols of this thesis: Secure Remote Password protocol (SRP) and Dragonfly. For each protocol, we justify our motivation to study its implementation and give an overview of its workflow and specification details.

Contents

3.1. A Long List of Protocols **30**
3.2. The Peculiar Case of SRP **31**
3.2.1. SRP Workflow 32
3.2.2. SRP Limitations 33
3.3. The Wide Adoption of Dragonfly **34**
3.3.1. Password Derivation 34
3.3.2. Commitment and Confirmation Phase 36
3.3.3. Simultaneous Authentication of Equals 36
3.3.4. Extensible Authentication Protocol 38

3.1. A Long List of Protocols

PAKE protocols essentially come in two variants. Firstly, so-called *balanced* or *symmetric* PAKE protocols, introduced in 1992 by Bellovin and Merritt [BM92] and formalized in [BPR00], require both parties to know the shared password in plaintext. This implies that a break into the server leaks the password for all users. Popular examples include Dragonfly [Har15] and CPace [HL19].

Secondly, so-called *augmented* or *asymmetric* PAKE protocols, for instance SRP [Wu98] and OPAQUE [JKX18], were introduced in [BM93] and later formalized in [BMP00; GMR06]. These protocols are designed to give the server access only to a password-*verifier*, and the password itself is only available to the client. Thus, upon a server compromise, an attacker is still forced to perform an exhaustive offline dictionary attack. Such additional protection makes the task significantly more difficult for the attacker and makes asymmetric PAKEs appealing for numerous industrial solutions.

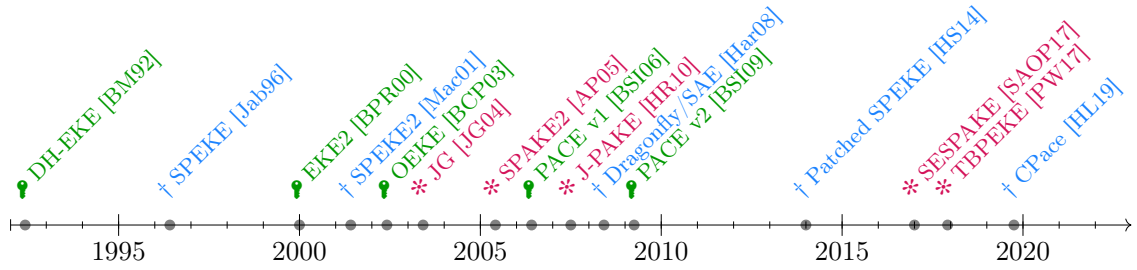
An Alternative Taxonomy. Besides the traditional, augmented/balanced classification, Hao and van Oorschot [HO22] introduced an alternative classification of different protocols based on their security model, assumptions, and construction approaches regarding password usage. As we will present in chapter 4, keeping internal password-related values secret is key to preserving the security properties of a PAKE. Hence, it is particularly relevant to identify how the password is derived and used along the protocol to pinpoint critical aspects in the implementation. Here, we will use a similar but less complete classification based on the three main ways password-derived values may be used. We identified the following usage:

- 🔑 as an encryption key;
- ✻ as a scalar (resp. exponent) for modular arithmetic over additive (resp. multiplicative) groups;
- † as input to derive a generator.

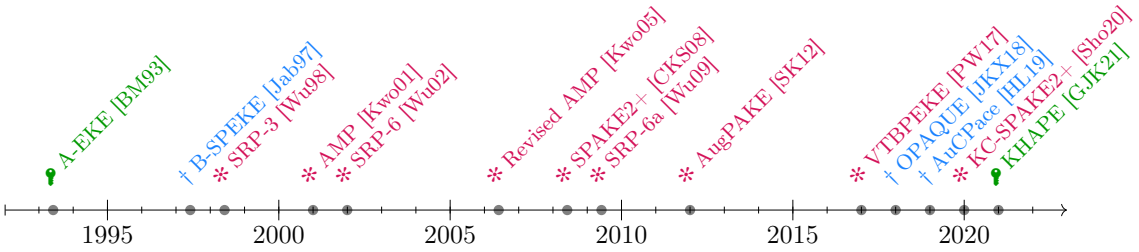
Numerous protocols, low deployment. Over the past decades, numerous protocols have been designed in response to three standardization processes. A recent study [HO22] provides a timeline of their proposal and the real-world application of the few that have been deployed on a large scale. An adaptation of this timeline is depicted in Figure 3.1, along with our aforementioned taxonomy.

Despite the considerable attention they received, they still suffer from slow adoption [EKSS09]. Indeed, the early protocols lacked important security properties, and the PAKE world was patent-encumbered for some time [BM92; Jab96]. This resulted in protocols, such as J-PAKE [HR10], that included additional complexity solely for patent circumvention, thereby making security proofs more complex or impractical. The availability of efficient implementations was also negatively impacted by this complexity.

However, interest has been renewed in PAKE recently with the expiration of the patents in 2018. Namely, the Crypto Forum Research Group (CFRG) started a competition in 2019 intending to recommend a small set of moderns PAKEs for the IETF community [CFR20]. The competition ended in March 2021, recommending OPAQUE [JKX18] as an augmented protocol, and CPace [HL19] for balanced. It is still early to measure the positive impact of this competition on PAKE adoption. However, it allowed the community to carefully review



(a) Balanced PAKE



(b) Augmented PAKE

Figure 3.1. – Simplified timeline of notorious PAKEs based on [HO22]. Balanced are represented in **(a)**, augmented in **(b)**. The color coding and symbols match the taxonomy on password usage we introduced: 🔑 as an encryption key; ✻ as a scalar (resp. exponent) for modular arithmetic over additive (resp. multiplicative) groups; † as input to derive a generator.

the various existing PAKEs concerning some specific criteria [Sch17]. Unfortunately, most of the reviews are based on the abstract design of the proposed candidates and not on their reference or deployed implementations. This is regrettable for two reasons. First, section 4.1 of the RFC 8125 [Sch17] explicitly requires that any PAKE shall protect against timing and side-channel attacks. Second, it is well established now that breaking cryptography is not merely a matter of theoretically breaking a cryptographic algorithm. Implementation pitfalls matter to guarantee that the running code does not leak any secret information.

3.2. The Peculiar Case of SRP

The SRP protocol is an augmented PAKE protocol introduced in 1998 [Wu98] with the primary goal to bypass existing patents and provide an open-source implementation. The protocol was standardized for the first time in 2000 in RFC 2944 [Wu00]. Since then, it has been updated multiple times to fix vulnerabilities, with SRP-6a being the latest version of the protocol [Wu09]. Nowadays, it is not part of the recommended protocols by the CFRG PAKE selection competition in 2020 [CFR20].

However, some argue that SRP is the most widely used and standardized protocol of its type, despite its lack of formal proof [HO22]. Its use in multiple commercial solutions reflects its popularity, as it is still deployed in 1Password (password manager) [Fil18], Apple iCloud (for credential recovery) [App21], Insomnia (IDE for HTTP-based API) [Ins20], AWS (Amazon Web Service in its cognito authentication extension) [Ass18], Apple Homekit (IoT

devices) [App20], ProtonMail (secure email system) [But16], etc. It draws its popularity from its patent-free definition, quite straightforward design, and the availability of open-source implementations with no restrictive licenses in many programming languages, including C#, .NET, Java, C/C++, Python, Go, JavaScript (and TypeScript), Rust, Erlang, and Ruby.

3.2.1. SRP Workflow

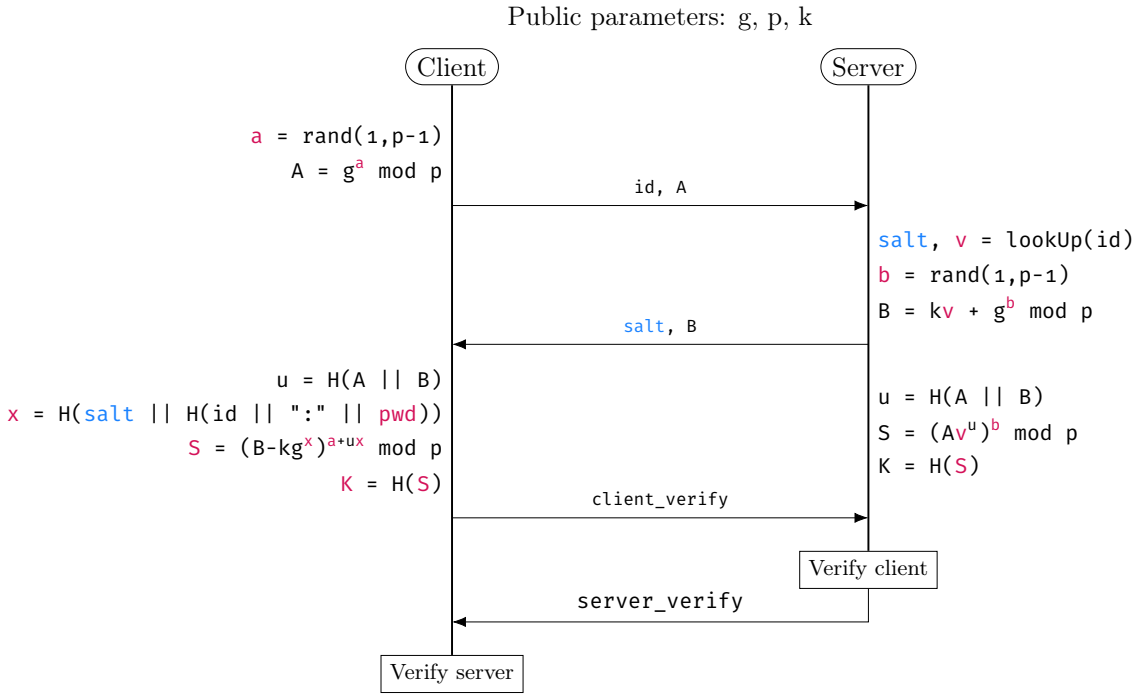


Figure 3.2. – SRP login workflow. Field parameters `g` and `p` are negotiated beforehand from a standardized set of parameters, and `k` is set to `H(g || p)`. Secret values are displayed in red, while the salt, as a sensitive value, is displayed in blue. We ommit the group negotiation frames.

SRP is a variant of Diffie-Hellman (DH), embedding a password-derived value to provide authentication. Hence, it benefits from the same security assumption, and its security relies on the Decisional Diffie-Hellman assumption. However, some parts of the protocol require additional assumptions compared to a classical DH: it leverages both an additive and multiplicative law. Hence, instead of a group, operations have to be performed on a *ring*. This particularity prevents any direct adaptation of the protocol to Elliptic Curve Cryptography (ECC).

Below, we describe the two phases of SRP: registration and login. Before, we discuss the assumptions made for its design:

- a field G of prime order p, which is a safe prime (*i.e.,* a number of the form $p = 2q + 1$, where q is also prime), and (multiplicative) generator g are publicly known;
- the particular field may be negotiated before the exchange or transmitted by the client;

- a public field-related parameter `k = H(g || p)` is computed by both entities upon the choice of the field, where `H` is any cryptographic hash function.

Registration. In SRP, the server never gets the password in plaintext. Instead, it stores the password in the form $(user_id, v, salt)$, where v is a password verifier and $salt$ is a random string generated at registration time. Let $x = H(salt \,||\, H(user_id \,||\, ":" \,||\, pwd))$, the verifier is generated as follows: $v = g^x \bmod p$. We stress that v is tied to the identifier, the salt and the password but no ephemeral value.

During this stage, the client should perform all computations and send the final tuple to the server over a secure channel, so an attacker cannot access the verifier. We assume this registration step has been completed successfully for all subsequent login attempts.

Login. Figure 3.2 summarizes SRP workflow, with secret information in red (the salt is a sensitive information, but not confidential). Before starting this phase, the client and the server agree upon the parameters g and p, as well as their digest $k = H(g \,||\, p)$. During login, first, the user generates a random number `a` uniformly chosen from $(1, p-1)$, computes $A = g^{\mathsf{a}} \bmod p$ and sends it to the server along with their identifier. Upon reception, the server looks up the user's password entry and fetches the corresponding verifier and salt. Then, the server generates a random number `b` uniformly chosen from $(1, p-1)$, computes $B = kv + g^{\mathsf{b}} \bmod p$, and sends both the salt and B to the user. Using the salt and the password, the client can recover the verifier $v = g^x \bmod p$. Now, both parties can compute the shared key K. Finally, they end up exchanging confirmation messages to verify the good proceedings of the protocol.

3.2.2. SRP Limitations

This part outlines some inherent flaws of SRP. These flaws are well known but do not result in any severe vulnerability.

Variation of SRP Settings. Since the first messages sent by the server are not authenticated, an attacker might modify them. This concerns the salt and the field parameters (g and p). Of course, such a modification would fail the session establishment.

Pre-computation Attacks. SRP as standardized does not protect from a leak of server-side data. Namely, an attacker can precompute a table of values based on the salt and a passwords dictionary. This implies that as soon as an attacker succeeds in compromising a server, they can instantly find a user's password without further effort. This is because anyone, including an attacker, can ask for any user's salt by providing their identifier and a random element. Such an attack is unfortunate, despite SRP being an augmented PAKE. To address the above threat, SRP would need a modern password-based key derivation function, such as scrypt, so pre-computations become expensive. Recently, *strong* asymmetric PAKEs (saPAKE) have been presented [JKX18], ensuring that an attacker is not able to perform such precomputation by the mean of oblivious computation of the verifier.

3.3. The Wide Adoption of Dragonfly

Dragonfly is a balanced PAKE, designed by Dan Harkins in 2008 [Har08]. It assumes a peer-to-peer scenario with no particular role attribution since the protocol follows the same workflow for both sides.

After stirring some controversy during the CFRG review process [Per13; Flu14], Dragonfly was properly described as RFC 7664 in 2015 [Har15]. Since then, Dragonfly and its variant, Simultaneous Authentication of Equals (SAE), have been officially endorsed and widely deployed as a key feature by IEEE 802.11 and Wi-Fi Protected Access (WPA)3 [IEE21; All18]. Despite being recently released, WPA3 already knows large adoption from major Wi-Fi providers and software. In particular, it has become mandatory for Wi-Fi certification since July 2020 [All20a]. Dragonfly is also used in informational RFC for Extensible Authentication Protocol (EAP) in pwd mode [ZH10] and TLS-pwd [Har19c]. In 2015, Lancrenon and Skrobot [LS15] provided a security proof of Dragonfly, including forward secrecy, in the BPR00 model [BPR00].

The handshake security relies on the discrete logarithm problem. Support for both multiplicative groups modulo a prime (MODP) and Elliptic Curve Cryptography over a prime field (ECP) is described in the standards [Har15; IEE21]. The exact operations of the handshake vary slightly depending on the underlying group. For the sake of brevity, we only consider ECP groups as the only mandatory group to support falls in this category. We stress that the standards explicitly mandate a curve with cofactor $h = 1$, excluding curves such as curve25519. As required by the specifications (section 12.4.4.1 of [IEE21]), we assume group 19 (corresponding to P256) as the default supported curve unless stated otherwise.

Thereafter, we adopt a traditional elliptic curve notation: E is the curve of order q defined over $\mathbb{F}_p$. Lowercase denotes scalars, and uppercase denotes group elements. Given the specification, we assume the equation to be in the short Weierstrass form $y^2 = x^3+ax+b \bmod p$, where a, b and p are curve-dependent and p is prime.

As illustrated in Figure 3.3, Dragonfly is broken into three main parts: (i) password derivation; (ii) password commitment; and (iii) confirmation.

3.3.1. Password Derivation

To begin with, both the sender and the receiver need to convert the shared password into a *group element*. For MODP groups, the easiest way is to hash the password into an integer modulo p. The process is less straightforward for ECP groups: we need to convert an integer into a point on the curve. Multiple algorithms exist and are currently being standardized [FSS+22]. Since 2020, two methods are part of the IEEE 802.11-2020 standard [IEE21].

Hunting-and-pecking. The default method, presented as the only available option in previous versions of the standard [IEE12; IEE16], is based on a probabilistic try-and-increment approach called *hunting-and-pecking*. This approach consists in hashing the password along with the identity of both peers and a counter. The counter is incremented, and the process is repeated until the computed value corresponds to a group element. This involves two steps. First, it converts the password into the x-coordinate of a point. Then, since two y-coordinates may be valid, only one is chosen based on the parity of some values. The pseudocode describing this process on ECP groups is summed-up in Listing 3.1.

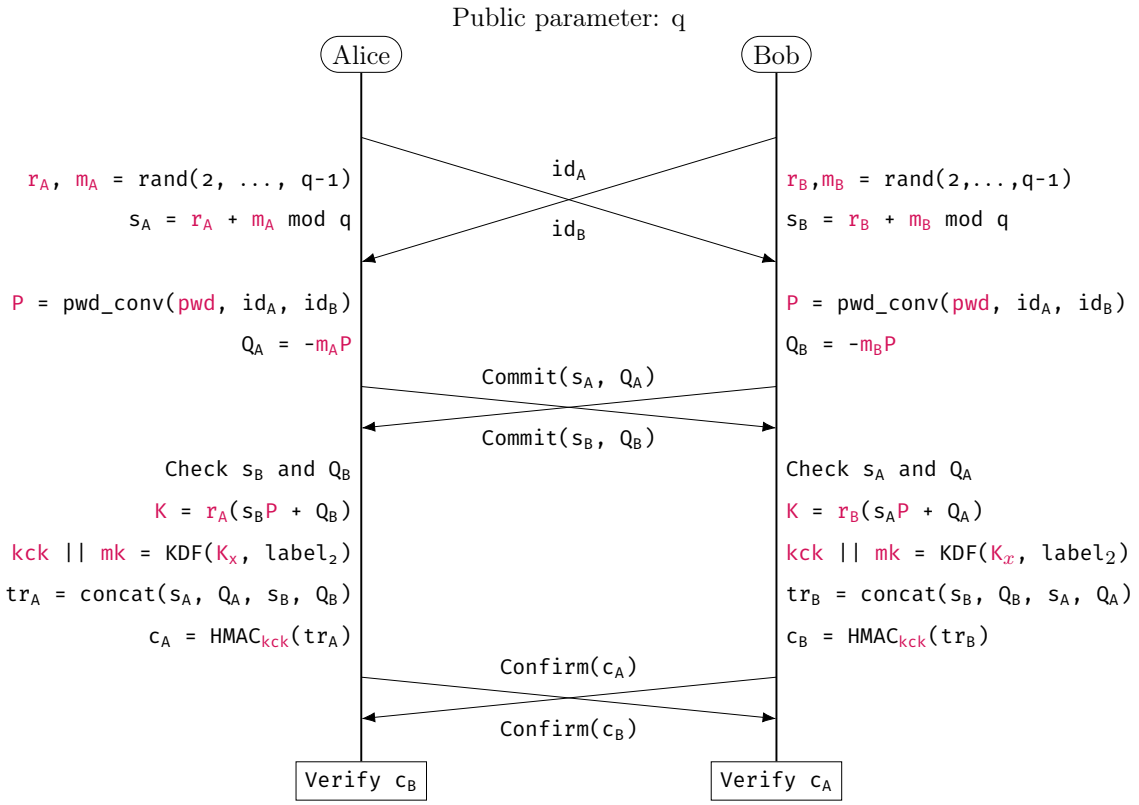


Figure 3.3. – Dragonfly handshake. Further details on the `pwd_conv` method can be found in Listing 3.1 and Listing 3.2. $label_2$ is a string that may vary along with the protocol instantiation. We use an additive group notation to fit the elliptic curve setting. Secret values are represented in red.

Some side channels have been identified along the standardization, leading to additional computations. Namely, the total number of iterations should be fixed so that the conversion performs the same number of iterations, no matter when the password is successfully converted. Each additional operation shall be performed on a random dummy value generated once on successful conversion (line 12 of Listing 3.1).

In addition, sensitive information may also leak when checking for the validity of the potential x-coordinate (Listing 3.1, line 8). This function basically computes $x^3 + ax + b$ and checks whether it is a square by computing its Legendre symbol (*i.e.,* a modular exponentiation). The protocol suggests to blind the computations by generating a random number for each test, squaring it, and multiplying it to the tested number. The result is then multiplied by a per-session random quadratic (non-)residue before computing the Legendre symbol. The square root is then computed once and for all at the end of the function.

Hash-to-element. Following the disclosure of *Dragonblood* [VR20] in 2019, both the Wi-Fi Alliance and EAP-pwd updated their standard to describe a password conversion method based on Simplified Shallue-van de Woestijne-Ulas (SSWU) [IEE21; Har19b]. This method is a deterministic, efficient, and easy-to-implement alternative to hunting-and-pecking. Readers can refer the pseudocode in Listing 3.2 for a high-level overview of the method, and to the

```
def hunting_and_pecking(pwd, macA, macB, ec):
    found, counter = False, 0
    A, B = max(macA, macB), min(macA, macB)
    while counter < k or not found:
        counter += 1
        seed = Hash(A || B || pwd || counter)
        x_cand = KDF(seed, label_1, ec.p)
        if is_x_coordinate(x_cand, ec):
            if not found:
                x, fmt, found = x_cand, seed, True
                # Not described in the RFC, but specified in SAE
                pwd = get_random_bytes(32)

    P = set_compressed_coord(x, 2 + (fmt & 1), ec)

    return P
```

Listing 3.1 – *Hunting-and-pecking* on ECP group, called `ec`, as used in WPA3. `pwd` is the secret password, `macA` and `macB` are the Media Access Control (MAC) of both peers, and `ec` is an elliptic curve. The value of `label_1` and `k` may vary along with the implementation, with a recommended value of `k`≥ 40.

specification for a detailed explanation.

We stress that this second method is not backward-compatible. Therefore, both peers need to agree on the password derivation method. The standard states that implementation should default to the original method (hunting-and-pecking) *"If the AP does not indicate support for the SAE hash-to-element in its Extended RSN Capabilities field or the SAE initiator does not set the status code to SAE_HASH_TO_ELEMENT in its SAE Commit message"* [IEE21].

3.3.2. Commitment and Confirmation Phase

Once the shared group element has been computed, both parties exchange a commit followed by a confirmation frame to conclude.

The commit frame is built with two values: a scalar $s_i = r_i + m_i \bmod q$, computed by adding two random nonces r_i, $m_i \in [2, q)$, and a commit element $Q_i = -m_i P$. When receiving this frame, a peer needs to check its validity (*i.e.,* whether $s_i \in [2, q)$) and Q_i belongs to the group).

In the confirmation phase, peers compute the master key, represented by the x-coordinate of K. This value is then derived into two subkeys using a Key Derivation Function (KDF): *kck* is a confirmation key, and *mk* is used as a master key for the subsequent exchanges. Using the confirmation key, an Hash-based Message Authentication Code (HMAC) is computed over the session transcript. The resulting tag is included in a *confirm* frame. The handshake succeeds only if all verifications end successfully.

3.3.3. Simultaneous Authentication of Equals

WPA3 uses a slight variant of Dragonfly, called SAE [IEE21]. The label values are fixed in this variant, and each peer is identified by its MAC address. Both peers are referred to as *stations*, or *STAs*. The SAE handshake is executed between the client and the Access Point (AP) to compute the Pairwise Master Key (PMK), called *mk* in Figure 3.3. Afterward, a classic WPA2 4-way handshake is performed with this PMK to derive fresh cryptographic

```
def hash2element(pwd, ssid, identifier , ec):
    len = olen(ec.p) + ⌊ olen(ec.p)/2 ⌋
    # extract seed material to get multiple random values
    seed = hkdf_extract(ssid, pwd || identifier )
    # expand the seed into a first random value
    pwd_value = hkdf_expand(seed, label_u1_P1, len)
    u1 = pwd_value mod ec.p
    P1 = SSWU(ec, u1)
    # expand the seed into a second random value
    pwd_value = hkdf_expand(seed, label_u2_P2, len)
    u2 = pwd_value mod ec.p
    P2 = SSWU(ec, u2)
    PT = P1 + P2
    # PT is stored until needed to generate a session-specific P
    return PT

def get_password_element(PT, macA, macB, ec):
    base = hkdf_extract(0, macA || macB)
    k = (base mod (ec.q − 1)) + 1
    return k × PT
```

Listing 3.2 – *Hash-to-Element* on ECP group, called `ec`, as used in WPA3. Capitalized variables denote points on the curve. The HKDF is based on the extract-then-expand paradigm. The first stage takes the input keying material and "extracts" a fixed-length seed. The second stage "expands" the seed into several additional pseudorandom values of chosen length. Note that PT is computed once, and then reused at session establishment to compute P.

materials. Because the entropy of this key is higher than in WPA2, the dictionary attack on the 4-way handshake is no longer relevant.

Since the first version of WPA3 issued in 2018, the original SAE protocol has been extended into multiple variants to provide new features and extend its security guarantees. As a result, we can differentiate three non-interoperable variants of the handshake, namely SAE, SAE-PT and SAE-PK.

SAE. The first version of WPA3 presented SAE [All18], with supports for the original hunting-and-pecking password conversion method only.

SAE-PT. In 2019, an update of the standard [IEE21] introduced SAE-PT as a mitigation for Dragonblood attacks by forcing the use of a deterministic password conversion method. SAE-PT mostly differs from SAE in the password conversion. A point, PT, is computed once per *(id, pwd)* tuple using the `hash2element` method described in Listing 3.2. This element is used on session establishment as the base of a scalar multiplication. The scalar is derived from the STAs' MAC addresses. With this release, the Wi-Fi Alliance aims to make the legacy SAE obsolete.

SAE-PK. Besides the side-channel vulnerabilities, a significant issue remained. WPA3 is likely to be deployed in hotspots, resulting in a model where multiple entities, including attackers, know the password. In this scenario, it would be beneficial to prevent malicious actors from creating a rogue AP and tricking the users. This issue is tackled in the third version of WPA3 [All20b], with the introduction of *SAE public key*. In this protocol variant,

the password is derived from the public key's fingerprint, and the knowledge of the private key is necessary to establish a session as an AP. The session establishment is similar but includes an additional step requiring the AP to sign a field to prove its knowledge of the private key.

3.3.4. Extensible Authentication Protocol

EAP is an authentication framework used in various protocols. It supports various authentications method, including a password-based authentication called EAP-pwd [ZH10]. This method rely on Dragonfly.

SAE and EAP-pwd being two variants of Dragonfly, they differ in a few points. Some are only instantiation details (values of some labels), while others have more impactful consequences on the workflow and the protocol's security. First, EAP-pwd standard [ZH10] does not mandate a constant number of iterations. Indeed, it exits the conversion loop once the password is successfully converted. However, since a constant number of iterations would not change the outcome of the conversion, some implementations include this side-channel mitigation regardless. Next, EAP-pwd does not benefit from the same symmetry as SAE: client and server are clearly defined. This distinction is highlighted by the server generating a random token for each new session. This token will be part of the information hashed at each iteration during the password conversion. Hence, while a password is always derived into the same element in SAE (for fixed identities), each EAP-pwd session ends up with a new group element due to the randomness brought by the token.

Side Channel in PAKE implementations 4

As with any cryptographic protocols, Password-Authenticated Key Exchange protocols (PAKE) may be subject to generic attacks. Such attacks include, but are not limited to, the following.

Reflection attacks. Since some PAKE (mainly balanced) adopt the same protocol on both sides, the value sent by a legitimate peer would be technically valid if the same peer received it. Hence, a Machine-in-the-Middle (MitM) attacker could intercept the value and send it back to the target, tricking it into providing the answer to its challenge. A protocol can easily address this type of attack by requiring peers to include an identifier in responses, for instance.

Replay attack. If the protocol relies on precomputed groups or fixed values, an attacker could passively intercept values and replay them in a future session to impersonate a user. This common issue is overcome at the protocol level by adding nonces.

Offline dictionary attack. This is a primary concern for PAKEs due to their particular properties. These attacks are typically addressed by carefully designing the protocol. One specific instance of these attacks is called two-for-one guess, where an attacker can eliminate two password candidates from a single session observation. Even this slight leakage is considered as breaking the security property.

In this chapter, we take a deeper look at the main pitfall of PAKE implementation. First, we discuss why any password-related leakage, as mild as it is, may be abused. Then, we demonstrate a generic technique to perform a dictionary partitioning attack given an arbitrary leakage. Once the theoretical attack is defined, we introduce a metric based on the signal-to-noise ratio to assess the practical efficiency of the attack. Finally, we discuss significant attacks targeting PAKE implementations within the last decades.

Contents

4.1. On the Danger of Leakage in PAKE 40
4.1.1. Leakage Aggregation: From Independent Bits to Fingerprint . . . 40
4.1.2. Leakage Impact: Dictionary Partitioning Attack 40
4.2. Assessing the Attack's Efficiency 41
4.2.1. Number of Traces to Reduce a Dictionary 41
4.2.2. Dictionary Independant Metric 42
4.3. Related Attacks on PAKEs 42

4.1. On the Danger of Leakage in PAKE

Common side-channel attacks on cryptographic implementation may be hindered by their need for a large amount of information on the secret to perform an attack. For instance, while leaking the Most Significant Bit (MSB) of Elliptic Curve Digital Signature Algorithm (ECDSA) nonces is devastating, leaking only a few bits from an RSA private key does not utterly break the security. This is because such attacks aim at solving the Discrete Logarithm Problem (DLP) from a partially known exponent. Similarly, sparse leakage may not suit the generic techniques for solving DLP. In such cases, the amount of information usually needs to compensate for the scarcity, for heuristic techniques to recover the entire key.

Here, we demonstrate why these constraints do not impact a leakage on password-related values. We describe a generic approach using leaked bits independently to build a password fingerprint and perform a dictionary partitioning attack on PAKEs, as done in [VR20].

4.1.1. Leakage Aggregation: From Independent Bits to Fingerprint

> **Reminder:** Cache-attacks, introduced in section 2.3, enables an attacker to spy on part of the execution.

Side-channel leakage samples are obtained by observing some execution of the PAKE. To illustrate our point, we consider a deterministic execution of the part leaking information. It may be the entire protocol (*i.e.,* given the same input, we get the same output) or a subset of the protocol (*i.e.,* an internal function).

This means that, except for masked computations, we can observe the same values for the same run. This is fortunate for the attacker since measurements may be noisy, thereby requiring several observations to obtain reliable leakage. The number of measurements needed for reliable observation and the amount of information in the leakage defines the *signal-to-noise ratio*.

Once the leakage is acquired, the attacker might need more data than one run can reveal to succeed in practice. Therefore, they may trigger a new leakage on the same secret value. This is possible only when the leaked secret-related value involves public parameters that can be maliciously modified.

Thus, given a spying process leaking ℓ bits of information on a secret value, we denote ℓ_i as the leakage related to a given execution, namely a fixed set of public parameters. Considering the leaky secret as the output of a random oracle seeded with both a password-related value and one or more public parameters, the collected leakages can be seen as independent information on the secret. Consequently, the overall leakage obtained by an attacker can be represented as follows: $\mathcal{C}_n = \{\ell_i, i = 1...n\}$. We say that $\mathcal{C}_n$ contains n traces. Ultimately, $\mathcal{C}_n$ may be used as a password fingerprint to perform a dictionary partitioning attack.

4.1.2. Leakage Impact: Dictionary Partitioning Attack

Once the fingerprint of the targeted password is leaked, the attacker can switch into an offline mode. Indeed, they apply the leaky operation to a dictionary of password candidates. Then, they discard each candidate whenever the leaked bits do not match the computed values. If the leakage was collected on an internal function, the attacker could drop any unnecessary computation to speed up the process.

The performance and success rate of the offline step are inherently linked to the dictionary size: the bigger the dictionary, the more candidates need to be tested. Depending on the fingerprint size, false positives are likely to happen if the total leakage $\ell = \log_2(\#dictionary)$ (since collision may happen).

4.2. Assessing the Attack's Efficiency

Considering this attack, two comparative analyses of attacks efficiency come to mind, based on: (i) the number of traces needed to recover a password in a given dictionary; (ii) the signal-to-noise ratio of the leakage.

The first element focuses on the comparison of the leakage from each trace. It is then relevant for theoretical comparison. The second, on the other hand, focuses on the efficiency of the practical attack since it considers the noise. We will use both metrics in the following chapters to compare our results to the relevant work in the literature.

4.2.1. Number of Traces to Reduce a Dictionary

An interesting question is how large the fingerprint should be to find the correct password uniquely. In other words, given a dictionary with L passwords, what is the smallest number of traces that eliminates all wrong candidates with high probability?

Intuitively, given a fingerprint $\mathcal{C}_n$, the probability that a password candidate does not match k leakages follows a binomial distribution, with p_i being the probability corresponding to each leakage. Thus, we can infer the probability that $L-1$ candidates are pruned given n traces. The smaller n, the more efficient the attack is.

Shannon defined the information content of an event as a function of its probability: $I(E) = -\log_2(p(E))$ [Sha48]. Let each leakage be represented by ℓ_i, the information it contains. Hence, the probability for a random password to leak the same information is $p_i = 2^{-\ell_i}$. Let Y_i be the random variable representing a password's success (1) or failure (0) to match the trace ℓ_i from the fingerprint. We got $Y_i = 1$ only if the password generates the same leakage for a particular set of parameters. Hence, the probability that a password candidate is eliminated given a trace ℓ_i is

$$\Pr[Y_i = 0] = 1 - \Pr[Y_i = 1] = 1 - p_i. \tag{4.1}$$

Let Y be the random variable describing the success or failure to match the entire fingerprint. The probability for a password to be pruned by at least one of the n traces is $\Pr[Y = 0]$. This is the sum of probabilities for the password to be pruned by any of the traces:

$$p_Y = \Pr[Y = 0] = \sum_{i=0}^{n-1} \left(\Pr[Y_i = 0] \cdot \prod_{j=0}^{i} (1 - \Pr[Y_j = 0]) \right). \tag{4.2}$$

A particular scenario where each collected leakage gives the same amount of information gives:

$$p_Y = \sum_{i=0}^{n-1} \left(\Pr[Y_i = 0] \cdot (1 - \Pr[Y_i = 0])^i \right). \tag{4.3}$$

Hence, the probability to prune d random (incorrect) passwords from a dictionary, given n traces, is p_Y^d.

From this result, we can predict how many traces we need to eliminate all wrong passwords from a dictionary with a given probability. Previous work on PAKE aimed at a probability of 0.95 [VR20], so we comply to this threshold. In practice, we do not need to remove all passwords from the dictionary. We only need to reduce it enough so that the remaining passwords can be tested in an active attack. Keeping more passwords in the dictionary would reduce the number of required traces.

A practical example of this result, extracted from [VR20] (section 7.2), reflects the case of a 1-bit leakage. In this case, an average of 29 traces are necessary to eliminate all invalid candidates from a dictionary of $d = 1.4 \cdot 10^7$ elements with a probability greater than 0.95 ($p_Y^d = 0.97$).

4.2.2. Dictionary Independant Metric

We propose to measure the efficiency of the offline attack, while not involving the dictionary size. Thus, we introduce $\mathcal{D}$, representing the signal-to-noise ratio. We define it as the ratio between the leakage for a fixed set of parameters (ℓ), and the number of measurements necessary to get a reliable leakage ($\mathcal{R}$). This is directly related to the number of required measurements to prune a given dictionary.

A perfect dictionary attack would allow an attacker to prune all invalid candidates by observing a single online authentication session. This would translate to $\mathcal{R} = 1$ and $\ell = \log_2(\#dictionary)$, hence $\mathcal{D} = \log_2(\#dictionary)$. Generally speaking, the bigger $\mathcal{D}$ is, the better the attack is.

4.3. Related Attacks on PAKEs

As we mentioned previously, numerous works study the logical flow of protocols or attempt to prove their security. For instance, Barry Jaspan [Jas96] showed that the initial assumption in EKE was insufficient to prevent offline dictionary attacks. Later, Bellare *et al.* [BBM00] proved that the core exchange of EKE is a secure protocol. Other protocols, such as SPEKE, have evolved in an attack-and-patch manner [Mac01; Zha04; HS14]. Thorough background on PAKE protocols and their history is given in chapter 8 of [BMS20].

More relevant to the protocols we study in this thesis, in 2014, Clarke and Hao [CH14] exposed a small subgroup attack due to a lack of validity verification.

Similarly, Secure Remote Password protocol (SRP) early design suffered from a "two-for-one" guess attack [Wu02]. SRP-6 adopts a peculiar design, requiring a ring instead of the classical group assumption, making security proofs more difficult. In [SLL+20], Sherman *et al.* model the addition ($g^b + v \bmod q$) as an encryption of the ephemeral key share with the verifier v. Under this assumption, they proceed to a formal analysis of SRP-3. They uncover a weakness allowing a malicious server to proceed to an authentication session without the client being involved. This may open to an escalation privilege. Besides this issue, no vulnerability was found, excluding known algebraic attacks.

However, very few attempt to assess their implementations or present practical attacks on deployed PAKEs.

Partitioning Oracle Attack on OPAQUE [LGR21]. In 2021, Len *et al.* presented an oracle attack on non-committing primitives. The attack consists of abusing the attacker's

ability to forge a ciphertext that decrypts without error, even if they do not have the secret key. By doing so, they can craft a ciphertext that decrypts successfully under many keys and abuse an oracle to perform a partitioning attack. This attack becomes particularly handy when a key is derived from a low entropy value, such as a password.

In this scenario, the attacker would build a set $\mathcal{S}$ of (password, key) candidates to test and forge a ciphertext that can be deciphered by half of the key. Then, for each query to the oracle, they would dismiss half of the set, leading to a password recovery in $\log_2(\#S)$ queries.

OPAQUE uses a password-related value as a group generator and encryption key for a symmetric cipher. Some early (pre-standardization) OPAQUE implementations are affected because they used a non-committing AEAD.

Miscellaneous issues in SRP impementations [Vol16]. A bug has been identified on cSRP (also affecting PySRP) and srpforjava implementations. Instead of computing $kv + g^b \bmod p$, the implementations calculate $kv + (g^b \bmod p)$, allowing to recover the most significant bits of v and hence perform a dictionary attack.

> **Reminder:** In SRP, the client computes a modular exponentiation with a password-related value x as an exponent, with $x = H(salt \mid\mid H(id \mid\mid ":" \mid\mid pwd))$ (section 3.2).

Power SCA on Apple's SRP [Rus21]. In 2021, Andy Russon [Rus21] described a Differential Power Analysis (DPA) attack affecting Apple's CoreCrypto library. They highlight a small leakage (the most significant bit may leak) in the euclidian splitting protection used to mask the modular exponentiation internal computation. This mild leak becomes significant if the attacker can repeat the measurements with the same exponent, which is the case for SRP. They also recover information during the computation of the verifier on the client side and use it to perform a dictionary attack.

> **Reminder:** Early versions of Dragonfly convert the password into an elliptic curve point using a probabilistic try-and-increment method called hunting-and-pecking (section 3.3).

Dragonblood attacks [VR20]. In 2020, Vanhoef and Ronen presented a collection of attacks against the recently standardized Wi-Fi Protected Access (WPA)3. The collection can be divided into downgrade attacks and weaknesses in the Dragonfly handshake. The latter covers attacks enabling Denial of Service (DoS) and password recovery. In particular, they unveiled two side-channel attacks.

First, on vulnerable implementations, the overall processing time by the Access Point (AP) depends on the password. For MODP groups, the processing time discrepancy is caused by additional iterations if a pseudo-random password-derived value exceeds the group's order. For some groups, the probability of such an event is significant, causing additional iterations for some passwords. In particular, for Brainpool curves, a password-derived value is compared to the order of the finite field $\mathbb{F}_p$, and the iteration is aborted if the value is greater than the order. This happens with negligible probability for NIST curves since p is close to a power of two, but not for Brainpool curves. Overall, some implementations exited the conversion loop as soon as a valid point coordinate was found or did not set a minimum number of iterations big enough to prevent this leakage. Hence, the processing time was directly related

to the number of iterations needed to convert the password. This lets a remote attacker learn information on the password.

Second, a cache attack leak information on the password conversion for elliptic curves if the control flow of the conversion loop is not secret-independent. The attacker can use a Flush+Reload gadget to learn whether or not the first iteration of the loop successfully converted the password. This is because a successful conversion induces additional processing, even if the loop continues with dummy operations.

> **Reminder:** SAE-PK (section 3.3) aims to prevent malicious users from setup a rogue AP. It achieves this property using a password derived from the legitimate AP public key. In this setting, the password is, roughly, a truncated hash of the public key, the Service Set Identifier (SSID), and a random modifier.

Time-Memory Trade-Off attacks on SAE-PK [Van22]. In 2022, Mathy Vanhoef demonstrated how to use pre-computation to improve attacks targeting a specific SSID. In the model of SAE-PK, the attacker wants to set a rogue AP, using the valid password. To this end, they need to either learn the private key or map the password to a different public key (corresponding to a private key they know).

The author showed how (for the lowest security settings) precomputed tables can efficiently map an SAE-PK password to a valid modifier and public key, if the private key is known. Initially, a precomputed table is linked to the password computed from the SSID. Hence, this attack is restricted to networks sharing the same SSID, or targeted attack where a network uses the same SSID for an extended period but refreshes the password regularly.

To further motivate the impact of a pre-computation attack, the author introduced a method to generate password collisions. They described how to create multiple networks with different SSIDs, but sharing the same password. This enables a table to be reused against different SSIDs.

5 Related Topics and Threat Model

Previous sections presented a high overview of the topic of this thesis and gave the necessary background to understand our contributions. However, both cryptography and Side-Channel Attack (SCA) are vast and evolving topics. As this work lies at the intersection of these domains, we do not aim to give a complete overview. Specifically, we focus on software-based microarchitectural side-channel attacks on Password-Authenticated Key Exchange protocols (PAKE). Below, we briefly describe some other aspects of these domains that we left aside and may interest the reader. Then we define the generic threat model that we consider by default in the following chapters.

Contents

5.1. Related Topics **46**
5.2. Threat Model **47**

5.1. Related Topics

Physical SCA. As mentioned previously, side-channel analysis is a recent and rapidly evolving domain that can be divided into many branches based on the origin of the leakage. Namely, we left aside any *physical* side-channel, introduced in [KJJ99]. They often require some physical access to the device or a way to monitor a physical aspect of the device, as demonstrated in [LKO+21] where they used a software feature to monitor power consumption. They are similar to software-based attacks in many aspects and mainly differ in the parameter the attacker measures. These include, but are not limited to, monitoring power consumption, electromagnetic emanation, sound or photonic emanation.

Fault Injection. While previous techniques rely on passively monitoring parameters along the execution, attackers may also actively disturb the victim's behavior. Based on fault injection, this more invasive branch of SCA consists of glitching the system during a specific computation and observing the output [BDL97; BS97]. By doing so, attackers may either disturb a particular operation, affecting the output in a well-chosen way, or bypass a check and evade a security mechanism. Various environment manipulations have been investigated to perform those fault injections, including voltage or clock glitching, or extreme temperature. Furthermore, these attacks are usually quite invasive and require physical access to the targeted device. Finally, it is worth mentioning the case of a software-based fault attack, such as Rowhammer [KDK+14], allowing attackers to affect the state of DRAM by frequently accessing specific memory locations.

Transient Execution Attacks. The recent discovery of Spectre [KHF+19] and Meltdown [LSG+18] has spawned an entirely new field of study focused on microarchitectural phenomenons, yet distinct from side-channel attacks. Although they are similar in many ways, transient execution attacks are crucially different since they directly leak the targeted data rather than indirect information. The principle is to exploit the attempts of the Central Processing Unit (CPU) to predict the execution of future instructions. While a misprediction would be reverted to let the execution unaffected, the state of some components is still affected by such behaviors. Considering an appropriate threat model, attackers may take advantage of this window to leak information on the accidentally loaded data, *e.g.,* using microarchitectural side channels. Consequently, transient execution attacks typically internally use a side-channel attack as a building block for transmission from the transient domain to the architectural domain.

Other attacks on cryptography. In this thesis, we focus on discovering attacks on a specific protocol. However, many attacks leverage implementation mistakes in other widespread protocols, such as Transport Layer Security (TLS) [BSY18; MBA+21]. Other recent contributions include studying a particular library and investigating its implementation of various primitives, Elliptic Curve Digital Signature Algorithm (ECDSA) being a common target. Vulnerabilities discovered in ubiquitous libraries are likely to affect numerous projects. One can refer to Cesar Pereida García *et al.* recent works on OpenSSL and NSS to get an idea of its constant-time state [GBY16; GB17; GHT+20; HGD+20]. Other works tackle core issues in the protocol design or primitive choice to break its security. Recently, it has been the case for Telegram [AMPS22], OpenPGP [BPH22], and MEGA [BHP22]. Finally, recent work

exposed that even if an implementation is secure, too broad specifications may still open to cross-configuration attacks when interacting with other implementations if the specifications are too broad and leave room for mistakes [FPS21].

5.2. Threat Model

Our attacks rely on the Flush+Reload attack [YF14], hence we need to consider the appropriate threat model. Here, we describe the generic assumption we make to exploit the vulnerabilities we discovered reliably. We may introduce other assumptions in the relevant chapters depending on the specific protocol we attack.

Since the operations required for this attack (cache access and eviction) do not rely on particular permissions, a common assumption is that the attacker deploys an unprivileged user-mode program on the targeted device. This spyware runs as a background task and records the CPU cache accesses to some specific functions. More precisely, this attack targets the Last Level Cache (LLC), assuming an inclusive cache structure, and the attacker needs to be able to execute the `clflush` instruction. These properties are commonly found on Intel processors.

As a side note, we stress that these assumptions do not hold for ARM processors (commonly used on iPhones, iPads, and recent Mac). Namely, the non-inclusiveness of the LLC and the absence of a user-land flush instruction make it harder to perform cross-core attacks. However, all these obstacles can be circumvented, as demonstrated in the work of Lipp *et al.* [LGS+16]. We encourage readers to refer to this article for further details on how to adapt the attack for the ARM architecture.

One might argue that this is a strong threat model. However, we insist that any unprivileged code can play the role of our spyware and, therefore, can be hidden into any third-party program running on the victim's computer. Moreover, previous paper suggests that a JavaScript code injection can remotely grant such memory access in web browser [OKSK15]. However, we did not investigate the effectiveness of our attack in such a context.

Part II.

Contributions

Password Recovery Attack Against SRP Implementations 6

In this chapter, we study a wide variety of Secure Remote Password protocol (SRP) implementations deployed in multiple industrial solutions. Instead of analyzing each project independently, we identify and exploit a common denominator of numerous softwares: OpenSSL. The chapter is organized as follows. In section 6.1, we set the bases of our contribution. We give our motivations to study SRP, set our threat model, and give an intuitive attack scenario. Then, in section 6.2, we identify the leakage vector in the client side of the OpenSSL SRP implementation and assess the theoretical average leakage in section 6.3. In section 6.4, we detail how to exploit this leakage in practice through a cache-based attack. We present a Proof of Concept (PoC)[1] in a real-world scenario, requiring only a single measurement to guess the secret password with high probability. In section 6.5, we demonstrate the large scope of our attack, that goes beyond OpenSSL and concerns widely deployed solutions, such as ProtonMail and Apple Homekit. Finally, in section 6.6, we describe the mitigations that we advised and the ones that were applied by affected projects.

Takeaway:

This contribution supports the following conclusions:

- ➤ Despite its deployment, reference SRP implementations are vulnerable to side-channel attacks.
- ➤ OpenSSL's strategy to provide insecure implementations by default is flawed and prone to error.
- ➤ Microarchitectural side-channel attacks are practical.

The appropriate background to grasp the details of our contribution is provided in:

- ➤ Chapter 2 for the background to understand the microarchitectural attack. Section 2.3 provides a detailed description of Flush+Reload.
- ➤ Chapter 3, especially section 3.2, to get an overview of SRP as a protocol.
- ➤ Chapter 4 for a description of the generic partitioning dictionary attack that we apply once we get the information on a password-related value.

[1] https://gitlab.inria.fr/ddealmei/poc-openssl-srp

Contents

6.1. Context and Motivation **53**
6.2. Attack on SRP Implementations **54**
6.2.1. OpenSSL Implementation 54
6.2.2. Vulnerability Details 57
6.2.3. Offline Dictionary Attack 60
6.2.4. Consequence of Password Recovery 60
6.3. Theoretical Amount of Leaked Information **61**
6.4. Experimentations **63**
6.4.1. Experimental Setup 64
6.4.2. Practical Attack on OpenSSL 64
6.4.3. Results 66
6.5. Practical Impact **68**
6.5.1. TLS-SRP using OpenSSL 69
6.5.2. Stanford Reference Implementation 70
6.5.3. Apple HomeKit Accessory Development Kit 71
6.5.4. LogMeIn Tools 72
6.5.5. ProtonMail Python Client 72
6.5.6. Big Numbers Libraries and SRP in the Wild 73
6.6. Mitigations **74**

6.1. Context and Motivation

> **Reminder:** Password-Authenticated Key Exchange protocols (PAKE) aim at establishing a secure and authenticated channel based on the shared knowledge of a password (chapter 3). In the case of asymmetric PAKE, such as SRP, the server does not know the password but a password-related value called *verifier* (section 3.2).

The case of SRP is quite peculiar. On the one hand, it has been highly disapproved by some of the crypto community [Gre18]. It was not even considered by the Crypto Forum Research Group (CFRG) PAKE competition. On the other hand, SRP is arguably among the most widely used PAKE. It is not only included in some amateur code on Github but also as security branding by various solutions (including 1Password [Fil18], Apple iCloud [App21], Insomnia [Ins20], AWS [Ass18], Apple Homekit [App20], and ProtonMail [But16]).

Despite being widely deployed, these implementations do not seem to have received any review for a long time. Therefore, looking deeper into the different SRP implementations becomes timely, as any identified vulnerability might lead to severe consequences. We aim to consider a modern threats model, especially cache-based attacks, to analyze SRP. Nevertheless, exploiting all the existing implementations is quite challenging for two main reasons. First, designing exploits that work for different programming languages is not straightforward. Second, SRP is used in diverse contexts, from mobile banking applications to video conferencing systems. Thus, any identified vulnerability would be hard to generalize.

In this chapter, we present another approach since we notice that most SRP implementations are rather similar. Indeed, we mainly focus on one reference implementation: OpenSSL's (in C). We show that the core part of the OpenSSL SRP has inspired at least 16 other projects in 5 programming languages (refer to Table 6.2 for the complete list). Moreover, these projects dynamically load the OpenSSL library to call its big numbers API. We underline that this implies that any vulnerability that can be identified in the OpenSSL code may constitute a generic attack vector for all these implementations, regardless of their programming language.

The SRP property that we attempt to attack is its resistance against offline dictionary attack (see chapter 4). In theory, an attacker eavesdropping on all messages within the authentication process can deduce no information about the shared password. We show that numerous SRP implementations do not satisfy such an important property.

> **Reminder:** As a default model, we consider the classical Flush+Reload model, defined in section 5.2. It assumes an unprivileged spyware running as a background task on the victim's computer.

Threat model. To increase the amount of leaked data, the attacker may repeat their measurements on a modified version of the session establishment (by changing the salt as described in subsection 3.2.2). We stress that this assumption is only needed to increase the efficiency of our attack. It turns out that we seldom require this; both theoretical (section 6.3) and experimental (subsection 6.4.3) amount of information that is leaked from a *single* session establishment allows us to reduce a large password dictionary entirely.

Attack Scenario. To illustrate our vulnerability, we propose the following attack scenario. Note that this only serves as an example and does not limit the scope of our work. We suppose a classical SRP use case, where a client tries to authenticate themselves to a server

using a password. The goal is to recover the client's password.

To this end, the attacker exploits a non-constant time execution during an insecure modular exponentiation caused by a mishandling of the constant-time flag. Here, the attacker leverages some particular properties of the victim's microarchitecture to monitor the execution and extract information about sensitive data. The attacker can then use this leaked information to perform an offline dictionary attack to recover the shared password with high efficiency. Once the attacker recovers the password, they can impersonate both server and client in all future sessions.

6.2. Attack on SRP Implementations

As mentioned previously, SRP has become widely deployed, and numerous implementations exist in various programming languages. Here, we provide an in-depth look at the OpenSSL implementation of SRP, describe why it is vulnerable, and explain how we can efficiently exploit it. In particular, OpenSSL provides support to TLS-SRP [TWMP07] through this implementation. Because of the popularity of OpenSSL, it is fair to assume that the code has inspired other implementations or is being reused as is (see section 6.5). We study OpenSSL implementation up to version 1.1.1i (included), which was the last version at the time of this contribution.

> **Reminder:** During SRP, the client performs a modular exponentiation with a password-related value. The exponent is computed based on the salt sent by the server, the user identifier, and the secret password fed to a hash function: $x = H(salt \,||\, H(id \,||\, ":" \,||\, pwd))$. The base of the exponentiation is denoted g, and the modulus p (section 3.2).

6.2.1. OpenSSL Implementation

> **Takeaway:** OpenSSL's implementation of the modular exponentiation is not secure by default. It is only secret-independent if the related flag is set in the big number's structure. SRP's implementation does not fill this criterion and uses an optimized square-and-multiply.

OpenSSL's Bignum. In many cryptographic operations, the size of the numbers to handle exceeds the size of a processor word (usually limited to 64 bits on modern architectures). Thus, libraries define a special structure called "big numbers". In OpenSSL, this structure, represented in Listing 6.1, contains a buffer of processor words (d), storing the data, the length of the buffer ($dmax$), the position of the word being processed (top), a sign flag (neg), and a flag describing any specific usage of the structure ($flags$). Of particular interest, following the work of Percival in 2005 [Per05], OpenSSL introduced a constant-time flag, `BN_FLG_CONSTTIME`, to identify sensitive big numbers. Nowadays, big numbers are checked for this flag in sensitive functions (*e.g.,* modular exponentiation) and processed accordingly.

OpenSSL's TLS-SRP. In TLS-SRP, OpenSSL uses SHA-1 as a hash function H, and supports all standardized groups in [TWMP07]. The code concerning SRP can be found in `crypto/srp/srp_lib.c` and `crypto/srp/srp_vfy.c`, and the standardized groups are hardcoded

```
1  struct bignum_st {
2      BN_ULONG *d;    /* Pointer to an array of 'BN_BITS2' bit chunks.
3                       * These chunks are organized in a least significant chunk
4                       * first order. */
5      int top;        /* Index of last used d +1. */
6      /* The next are internal book keeping for bn_expand. */
7      int dmax;       /* Size of the d array. */
8      int neg;        /* one if the number is negative */
9      int flags;
10 };
11
12 typedef struct bignum_st BIGNUM;
```

Listing 6.1 – OpenSSL representation of a big number.

```
1  # g and p are group parameters, x is the exponent. B is the server's key share
2  # a is the client's private share, and u = H( g^a mod p || B)
3  def SRP_Calc_client_key(p, g, B, x, a, u):
4      # Compute the verifier v = g^x mod p
5      v = BN_mod_exp(g, x, p)
6
7      # Compute k = H(p || g)
8      k = srp_Calc_k(p, g)
9
10     # Compute the base base = B - kv mod p
11     tmp = BN_mod_mul(v, k, p)
12     base = BN_mod_sub(B, tmp, p)
13
14     # Compute the exponent exp = a + ux
15     tmp = BN_mul(u, x)
16     exp = BN_add(a, tmp)
17
18     # Compute the shared S = base^exp mod p = (B - g^x)^(a+ux) mod p
19     S = BN_mod_exp(base, exp, p)
20
21     return S
```

Listing 6.2 – OpenSSL v1.1.1i implementation of the shared key computation by the client. A python-like syntax has been used to save space and add clarity. Variable names have been replaced to fit the protocol description in section 3.2.

as big numbers in `crypto/bn/bn_srp.c`. In particular, the client-side function that computes the shared key (and the verifier v), `SRP_Calc_client_key`, is illustrated in Listing 6.2. We notice that all big numbers involved in the verifier (line 5) have default (non-secure) flags set. In particular, the `BN_FLG_CONSTTIME` flag is not set.

OpenSSL's modular exponentiation. The function `BN_mod_exp` only calls the secret-independent exponentiation as a fallback if a secure flag is set on one of the big numbers (base, exponent, or modulus), or if the base is too large for the computation to be optimized (more than one processor word). Otherwise, the optimized modular exponentiation `BN_mod_exp_mont_word` is called (Listing 6.3). All functions are defined in defined in `crypto/bn/bn_exp.c`.

This fast exponentiation uses a square-and-multiply (from most to least significant bit), with Montgomery multiplication [Mon85]. The leading zero bytes are ignored for maximum

```
1  def BN_mod_exp_mont_word(g, x, p):
2      nbits = BN_nb_bits(x)
3
4      # w is the word accumulator, used to make fast computations
5      # Set it to g since the exponentiation starts after the first bit to 1
6      w = g
7
8      # Do the optimized square and multiply on processor word only, until w overflows
9      for b in range(nbits-2, -1, -1):
10         # Square at every step
11         next_w = w*w
12         if (next_w / w) != w: # overflow
13             w = 1
14             break
15         w = next_w
16         # Multiply if the bit is 1
17         if BN_is_bit_set(x, b):
18             next_w = w * g;
19             if (next_w / g) != w: # overflow
20                 w = g
21                 break
22             w = next_w
23
24     # Once the word accumulator overflowed, we need a big number to
25     # store its Montgomery representation as a "long-term" accumulator.
26     r = BN_to_montgomery_word(w, p)
27
28     # Then get back to the square-and-multiply, with the additional
29     # constraint to update the long-term accumulator r every time the
30     # word accumulator overflows.
31     for bit in range(b, -1, -1):
32         # Square at every step on the processor word
33         next_w = w * w;
34         if (next_w / w) != w:
35             # If it overflows, update r, and reset the word
36             r = BN_mod_mul_word(r, w, p)
37             next_w = 1
38         w = next_w;
39
40         # We update the Montgomery representation by squaring at each step
41         r = BN_mod_mul_montgomery(r, r, p)
42
43         # If the bit is set, we multiply by g
44         if BN_is_bit_set(x, bit):
45             next_w = w * g;
46             if (next_w / g) != w:
47                 # If it overflows, update r, and reset the word
48                 r = BN_mod_mul_word(r, w, p)
49                 next_w = g;
50             w = next_w;
51
52     if w != 1:
53         r = BN_mod_mul_word(r, w, p)
54
55     return BN_from_montgomery(r)
```

Listing 6.3 – Modular exponentiation as performed by OpenSSL. Code has been reorganized and uses a python-like syntax to be more readable.

efficiency, and the exponentiation only starts after the first Most Significant Bit (MSB). The main optimization consists in avoiding big number computations and using a simple processor word w as often as possible. The word w (whose bit-size is defined by the processor architecture) is used as an accumulator to perform fast multiplications and squaring until its overflow (first loop from line 9 to line 22). Once it overflows, a big number r (long-term accumulator) is initialized with the Montgomery representation of w. In the end, the big number r will represent the result of the modular exponentiation. The word accumulator w is reset to 1 (if it overflowed on squaring) or g (if it overflowed on multiplication). For the remaining iterations (line 31 to line 50), the process is fairly similar: fast operations are performed on the accumulator, and every time it overflows, the big number is updated accordingly ($r = r \times w$). However, in this second part, the big number r is also squared at each iteration (line 41) to keep it updated, making the overall iteration longer.

6.2.2. Vulnerability Details

> **Takeaway:** Despite operations faster than the temporal resolution of Flush+Reload, an attacker can recognize patterns in the exponent (leading to a partitioning oracle attack on SRP). Based on the pattern they observe, they can extract a variable amount of information on the exponent.

Targeting the modular exponentiation. Since no constant-time flag is set, and the base g is either 2, 5, or 19, the optimized exponentiation is performed during the verifier computation. In particular, the secret exponent $x = H(salt \mid\mid H(id \mid\mid ":" \mid\mid pwd))$ is directly related to the password, which implies that recovering information on the exponent would provide the attacker with enough information to perform an offline dictionary attack reducing the set of possible passwords. Information is leaked in several ways.

First, skipping the leading zero bits makes the overall execution time depends on $\log_2(x)$: the number of required iterations leaks the number of leading zeros in the exponent.

Moreover, we note that square-and-multiply might also leak information since the execution of each iteration depends on the value of the bit being processed. Unfortunately for the attacker, performing these operations on processor words makes them very fast. Consequently, it is hard to distinguish them in practice, even using high-resolution cache attacks, such as Flush+Reload. Hence, recovering information during the first loop (line 9 to line 22 of Listing 6.3) is hard and error-prone.

However, starting from the first overflow on the accumulator w, the Montgomery representation is involved, making each iteration significantly longer, therefore, more distinguishable. Hence, an attacker can exploit the Flush+Reload attack to monitor the memory line corresponding to the squaring of r at each iteration (call to `BN_mod_mul_montgomery` on line 41 of Listing 6.3). This function being only called once every iteration, knowing when it is called allows the attacker to distinguish iterations from one another. Still, they cannot distinguish whether the bit value is 0 or 1 because the instructions are processed faster than the temporal resolution of Flush+Reload.

However, if w overflows, an additional call to `BN_mod_mul_word` is performed (line 36 and line 48 of Listing 6.3), making the particular iteration longer. Therefore, the delay between two different iterations (or two calls to `BN_mod_mul_montgomery`) will be longer. This delay can be increased by performing a Performance Degradation Attack (PDA) on `BN_mod_mul_word` to

make overflow iterations even longer to execute, hence easier to identify.

Consequently, an attacker can distinguish iterations and determine whether an overflow occurs. Interestingly, the number of iterations between two overflows is directly related to the binary sequence of the exponent being processed. Thus, the attacker can guess some bit patterns inside the exponent, which we detail below.

Exploiting the Leakage. The possible bit patterns we can observe between two overflows of the accumulator w depend on two parameters: the generator g and the bit size of the processor word in a particular architecture. To demonstrate the attack, without loss of generality, we fix the values for both parameters in the following sections. In order to meet the experimental setup we were using, we will consider w as a 64-bit word, which is the case for any x64 processor.

Choice of the Generator. The choice of the generator is motivated by two major criteria. First, we consider the security/performance ratio offered by all standardized groups: groups 1024 and 1536 are discouraged because of their low security. Similarly, group 2048 only provides satisfying security guarantees until 2030 [BD15]. Hence, higher order groups are more likely to be used. Finally, groups 6144 and 8192 may cause performance issues for some usage and only bring unnecessary security. Hence, a reasonable choice would be to use either group 3072 or 4096, both using a generator $g = 5$.

The following criterion is only valid from an attacker's perspective: an attacker would prefer a generator containing as many recognizable patterns as possible. Indeed, using $g = 19$ allows us to extract more information on average using the patterns alone. However, we show that $g = 5$ is the most interesting candidate when combining the pattern-related leak and the leak from the leading bits. Additionally, it is more likely to be used in practice ($g = 19$ being reserved for group 8192). Henceforth, we fix the generator to $g = 5$ in the following sections. We stress that for other values of g, the same reasoning can be applied but results in different possible patterns and a lower average leak. For the sake of completeness, we provide a full study for all standard values of g in section 6.3.

Identifying bit patterns. Assume we start with a fresh accumulator $w = g$, meaning we either start the exponentiation or we overflowed and processed a bit set to 1 (otherwise $w = 1$, and processing bits to 0 would not modify the value w).

Now, we can generate all combinations leading to an overflow on w by applying the square-and-multiply on different bit sequences, and stop when the result is greater than 2^{64}. Thus, we identified two cases: (i) if the first two bits are 1, an overflow occurs after three more iterations (for 111 and 110); (ii) otherwise, an overflow occurs after four iterations.

Let ℓ be the maximum bit length of the exponent, and let $T = \{v, V\}^{\ell}$ be the trace representing the list of iterations corresponding to the exponentiation. Here, v denotes a classic iteration (processing either a 0 or a 1), and V denotes an iteration where an overflow of w occurs (*i.e.,* an iteration where line 36 or line 48 of Listing 6.3 is executed). We show in section 6.4 how an attacker can recover such a trace with Flush+Reload.

Let $b \in \{0, 1\}$ denote a bit of unknown value, and t be a chunk of the trace representing the operations between two overflows. Then, we can convert our trace T into a set of possible values by extracting information on each chunk t as follow:

- A chunk $t = Vvv$, $t = Vv$ or $t = V$ is impossible, since at least 4 operations are needed to get the overflow.
- If $t = Vvvv$:
  - We know that $w = g$ at the first iteration following the overflow (otherwise, an additional iteration is needed to overflow), hence the first bit processed is a 1: $t = 1bbb$.
  - An overflow in 3 iterations with $w = g$ is only possible if the next two bits are 11. We end up with $t = 111b$.
- If $t = Vvvvv$, we cannot recover the exact bit values, but we can reduce the set of possible values:
  - If $V = 1$, t is in the set $\{110bb,\ 10bbb\}$. Otherwise, the accumulator would have overflowed one iteration before.
  - Otherwise, we meet the previous condition with a leading zero and get $t = 0111b$.

  We note $yyyy \in \{110b,\ 10bb,\ 0111\}$ the bits that take one of these seven possible patterns. Hence, $t = yyyyb$
- If $t = Vv...v$ with $len(t) = |t| = 5$, the firsts $|t| - 5$ bits of t (including the leading V) are 0. Hence $t = 00..0yyyyb$

On the last chunk of a trace, we can only guess that leading bits are zeros if its length is greater than five, because we do not know when the accumulator would have overflowed.

Bits Pattern Parser. We designed an automated parser to interpret the information leaked by our spy process. Our parser produces guesses for as many bits values as possible.

```
Trace:           Vvvvv Vvvv Vvvvvv Vvvvv Vvvvv Vvvvv Vvvv
Interpretation:  yyyyb 111b 0yyyyb yyyyb yyyyb yyyyb bbbb
```

Figure 6.1. – Example of trace interpretation. The first line is the original trace, recovered using Flush+Reload, and the second one is the interpretation.

Figure 6.1 presents an example, extracted from a real measurement, of how we can interpret a trace to recover part of the bits. Strictly speaking, in our example, we can only recover 4 bits. However, each $yyyy$ can only take seven possible values (*i.e.,* 110b, 10bb, and 0111), which significantly limits the possible values of the exponent. As mentioned previously, we cannot deduce information on the last chunk.

> **Reminder:** When a password-related value is computed from an unprotected public value, an attacker may alter the public value and repeat their measurement to aggregate information on the password (subsection 4.1.1).
> In SRP, the salt is public and transmitted unprotected in the first frame from the server (section 3.2).

Aggregating information. If the information recovered from a single session establishment does not allow for pruning enough passwords from the dictionary, the attacker can exploit a second session and change the salt transmitted by the server. Thus, they get different execution traces corresponding to the same password. Next, we discuss the amount of leaked data from a single session both theoretically (section 6.3) and experimentally (section 6.4).

6.2.3. Offline Dictionary Attack

> **Reminder:** Since the secret value is only defined by the password and some public values, they can compute various candidates for x and check whether the outcomes would match the recovered part (section 3.2 and section 4.2).

The above leakage allows an attacker to recover a significant part of x. If multiple passwords match the pattern, the attack is repeated by modifying the salt (which is sent unprotected by the server), and thus a new x corresponding to the same password is observed.

Since a cryptographic hash function is expected to behave like a random oracle, the digest corresponding to an invalid password will match the k bits of our trace with probability 2^{-k}. Therefore, an attacker recovering k bits of information on an exponent can expect to eliminate all candidates in a dictionary of 2^k entries.

Precomputations are still possible. Indeed, hashing password candidates can be performed before the attack. However, this needs to be done for each client username and each salt. In order to reduce the effort of precomputations, the attacker can force the use of a fixed salt and only varies username. These precomputations are fast, considering there are only two hashes to compute on small data. Depending on the specific hash function, hardware acceleration may be available to speed up the process.

6.2.4. Consequence of Password Recovery

> **Reminder:** SRP is an asymmetric PAKE, so the client knows the password, but the server only knows a *verifier* (section 3.2).

Assuming the attacker knows the password, they can: (i) impersonate the client; (ii) impersonate the server (since the attacker can compute the verifier from the password).

The required measurements are as fast as the session establishment, so the dictionary reduction dominates the overall complexity. If an attacker has significant computational power, it can be performed on the fly. Hence, the attacker could gain a full Machine-in-the-Middle (MitM) position and passively or actively process all data transmitted through the secure channel. The attack is still interesting even when it might take some time because of bigger dictionaries or more restricted computational power. Indeed, recovering the password after the sessions still allows the attacker to impersonate the client and/or gain active MitM position in all future sessions.

We give an insight on the practical cost of our attack for various dictionaries in subsection 6.4.3.

6.3. Theoretical Amount of Leaked Information

> **Takeaway:** The generator $g = 5$ gives the most variety of patterns, and offers the best average leakage: $\mathcal{L}(\ell) \approx 0.4 \cdot \ell + 1.2$, ℓ being the bit-size of the hash function's output. Using a hash function with larger output would only increase the exponent size, thus leaking more information.

The value of the exponent x is the output of a cryptographic hash function of bit-length ℓ. In this context, we can compute the average leakage based on the length ℓ and the probability of meeting the different patterns in a uniformly distributed bit string. The potential patterns and their probability depend on the group and, more specifically, its generator g.

Our spy process does not build a trace for the leading bits. The reason is twofold. First, the exponentiation starts at the MSB of the exponent. Second, the first iterations are fast and hardly observable since their corresponding computations are performed without Montgomery representation.

Next, on the remaining bits, we can identify patterns as described in subsection 6.2.2. Let L be the random variable corresponding to the information leaked by a pattern. Given a trace of k bits, we can estimate the average length $\mathbb{E}[N]$ of a pattern and the average information $\mathbb{E}[L]$ leaked by each pattern to get the information we can recover:

$$\mathcal{L}_{patterns}(k) = \mathbb{E}[L] \times \frac{k}{\mathbb{E}[N]}.$$

By combining these two sources of leakage, we get the total leakage based on the bit-length ℓ of the exponent.

Below, we detail how we estimated which one of the defined generators offers the best average leak for an attacker.

We start by implementing a script to produce all possible bit patterns leading to an overflow of the accumulator. Assuming we start with a fresh accumulator $w = g$, we notice that: (i) $g = 2$ implies that w always overflows after five more iterations, (ii) $g = 5$ implies that w can overflow after three or four iterations, and (iii) $g = 19$ implies that w always overflows after three iterations.

Let k denote the number of iterations in the trace, and ℓ be the bit-length of the exponent (including zero bits needed to match the length defined in the implementation specification).

> **Reminder:** The information given by an event E of probability p_E can be computed as $I(E) = -\log_2(p_E)$ (section 4.2).

Generator g = 5

Leaks from the leading bits. On average, we can expect $\sum_{i=0}^{\ell} i2^{-i-1} \approx 1$ leading zero bits. Then, since the square-and-multiply loop starts after the first bit set to one, the accumulator is initialized to $w = g$, and three patterns are possible before the first overflow: (i) $111b$ with probability 2^{-2}, (ii) $110bb$ with probability 2^{-2}, (iii) $10bbb$ with probability 2^{-1}. The average length of the first undetected pattern is, therefore, 4.75.

Hence, for a random exponent, we expect to detect $\ell - 5.75$ iterations on average. In

addition, the average information we can recover from the leading bits is expressed as

$$\mathcal{L}_{lead} = \sum_{i=0}^{\ell} i \cdot 2^{-i-1} + 3 \cdot 2^{-2} + 3 \cdot 2^{-2} + 2 \cdot 2^{-1} \approx 3.5.$$

Assuming ℓ is large enough, its impact on the leakage through the leading bits is negligible.

Leaks from the patterns. Next, we can determine the leakage by identifying each pattern as described in subsection 6.2.2. Let $\lambda \geq 4$ denote the number of iterations in the pattern, and p_λ the probability of meeting a pattern of this size in a random string.

- $\lambda = 4$: the three first bits are set to 1, which occurs with probability $p_4 = 2^{-3}$, thereby leaking 3 bits of information.
- $\lambda = 5$: the pattern is in the set $\{10bbb,\ 110bb,\ 0111b\}$. This occurs with probability $p_5 = 2^{-2} + 2^{-3} + 2^{-4}$ and leaks approximately 1.2 bits of information.
- $\lambda \geq 6$: the $\lambda - 5$ firsts bits of the pattern are 0. This occurs with probability $p_\lambda = 2^{-\lambda+5}$ and leaks $\lambda - 5$ bits of information. Furthermore, the last 5 bits verify the previous condition, meaning we get 1.2 additional bits of information.

The last two cases can be combined to get a pattern with $\lambda \geq 5$ bits and probability $p_\lambda = \left(2^{-2} + 2^{-3} + 2^{-4}\right) \cdot 2^{-(\lambda-5)}$, leaking $\lambda - 5 + 1.2$ bits of information.

We get the average leak from a pattern with

$$\mathbb{E}[L] = \sum_{i=4}^{\ell} p_i I(p_i) = p_4 I(p_4) + \sum_{i=5}^{\ell} p_i I(p_i) \approx 2.29.$$

Let N be the random variable corresponding to the length of a pattern. The average length of a pattern is:

$$\mathbb{E}[N] = \sum_{i=4}^{\ell} i p_i = 4p_4 + \sum_{i=5}^{\ell} i p_i \approx 5.75.$$

Finally, we can get the amount of leaked information by estimating the number of patterns we can expect based on the length k of the bit string we are considering:

$$\mathcal{L}_{patterns}(k) = \mathbb{E}[L] \times \frac{k}{\mathbb{E}[N]} \approx 0.4 \cdot k.$$

Hence, the average leakage from an ℓ-bit random exponent is

$$\mathcal{L}(\ell) = \mathcal{L}_{lead} + \mathcal{L}_{patterns}(\ell - 5.75) \approx 0.4 \cdot \ell + 1.2.$$

Generator g = 2

Since all overflows occur after five iterations once $w = g$, the pattern in our trace looks like $t = 0 \ldots 01bbbbb$, with $|t| = \lambda > 5$, with probability $p_\lambda = 2^{-\lambda+6} \times 2^{-1}$, leaking $I(p_\lambda) = \lambda - 5$ bits of information. The average information leak per pattern is, therefore

$$\mathbb{E}[L] = \sum_{i=6}^{\ell} p_i \times I(p_i) = \sum_{i=6}^{\ell} 2^{-i+5} \times (i - 5) \approx 2.$$

Following the same reasoning, we can get the average pattern length $\mathbb{E}[N]$ and deduce the expected leakage for an ℓ-bit exponent:

$$\mathbb{E}[N] = \sum_{i=6}^{\ell} p_i \times i = \sum_{i=6}^{\ell} 2^{-i+5} \times i \approx 7.$$

Hence, the average leakage from patterns is equal to

$$\mathcal{L}_{patterns}(k) = \mathbb{E}[L] \times \frac{k}{\mathbb{E}[N]} \approx \frac{2}{7}k$$

The average number of leading zero bits does not depend on the generator, but the information leaked on the first pattern leading to the first overflow does. Here, we only have one possible pattern length, which does not leak any information, therefore:

$$\mathcal{L}(\ell) = 1 + \mathcal{L}_{patterns}(\ell - 7) \approx 0.29 \cdot \ell - 1.$$

Generator g = 19

Computations are very similar for $g = 19$ except all overflows occur after three iterations once $w = g$, the pattern in our trace looks like $t = 0 \ldots 01bbb$, with $|t| = \lambda > 3$, with probability $p_\lambda = 2^{-\lambda+4} \times 2^{-1}$, leaking $I(p_\lambda) = \lambda - 3$ bits of information. The average information leak per pattern is, therefore

$$\mathbb{E}[L] = \sum_{i=4}^{\ell} p_i \times I(p_i) = \sum_{i=4}^{\ell} 2^{-i+3} \times (i - 3) \approx 2.$$

Following the same reasoning, we can get the average pattern length $\mathbb{E}[N]$ and deduce the expected leakage for an ℓ-bit exponent:

$$\mathbb{E}[N] = \sum_{i=4}^{\ell} p_i \times i = \sum_{i=4}^{\ell} 2^{-i+3} \times i \approx 5.$$

Hence, the average leakage from patterns is equal to

$$\mathcal{L}_{patterns}(\ell) = \mathbb{E}[L] \times \frac{\ell}{\mathbb{E}[N]} \approx \frac{2}{5}\ell.$$

Similarly, the average total leakage is:

$$\mathcal{L}(\ell) = 1 + \mathcal{L}_{patterns}(\ell - 4) = 0.4 \cdot \ell - 0.6.$$

6.4. Experimentations

Takeaway: A practical implementation of our attack gives a 96.9% success rate from a single measurement, the remaining 3.1% being undoubtedly identifiable as non-conclusive. This single trace leaks 63.9 bits from a 160-bit exponent on average, enabling the reduction of large dictionaries.

6.4.1. Experimental Setup

All experiments were performed on a Dell XPS13 7390 running on Ubuntu 20.04, kernel 5.4.0-58, with an Intel(R) Core(TM) i7-10510U and 16 GB of Random Access Memory (RAM). Binaries were compiled with gcc version 9.3.0-17ubuntu1 20.04 using their build script with the default configuration (optimization included).

OpenSSL's security team successfully reproduced our experiments in a Docker container on the same hardware. This Docker includes all elements needed to reproduce our experiments, from data acquisition with the victim program to the dictionary attack recovering the password. It is available in our GitHub repository[2].

We used the FR-trace program of Mastik v0.02 [Yar16] as a spy process performing both Flush+Reload and PDA. We ran all tests on OpenSSL version 1.1.1h, but a code analysis suggests that all versions of OpenSSL up to v1.1.1i (included) are vulnerable.

6.4.2. Practical Attack on OpenSSL

> **Reminder:** Early messages of SRP, including group parameters and salt, are not protected. Hence, an attacker can define the group (within the standardized group) and choose the salt. This is covered in the known flaws of SRP (subsection 3.2.2).

Trace collection. To demonstrate our attack, we used a simple C program with passwords drawn randomly from a dictionary to compute the verifier with the OpenSSL function `SRP_compute_client_key`. As motivated earlier, we used the generator $g = 5$. Specifically, we worked with the 6144-bit MODP group, described in [TWMP07], so the client uses the largest modulus available with this generator. Other parameters (*e.g.,* the salt) are randomly generated to avoid the setup of a server.

Once the spyware runs as a background process, the execution of our victim program will produce an execution trace. Figure 6.2 depicts a part of such a trace. The red line represents the threshold we used during the Flush+Reload: any value below this threshold represents an execution of the monitored code (*i.e.,* a new iteration). Any value above the threshold means that the victim performs some other computation. In this trace sample, we can identify four groups of iterations, separated by a significant time gap (horizontal gap). This gap identifies an overflow: the iteration is longer because of the additional operations and the delay caused by the PDA. We note that trace collection is as fast as an SRP session establishment.

Trace interpretation. From the data acquired by the spy process, we need to extract exploitable information on the exponent. To avoid the cumbersome task of manually interpreting the trace, we implemented a parser script taking a raw measurement and outputting the list of pattern lengths composing the exponent.

The parser works as follows: after sanitizing the raw trace (by removing trailing measurement detecting no activity, and the trace header containing system information), we applied heuristic methods to produce a more reliable trace.

We proceed to the interpretation in two steps. First, outlier values are removed, and impossible patterns close to the minimal pattern size are rounded up (these errors are usually

[2] https://gitlab.inria.fr/ddealmei/poc-openssl-srp

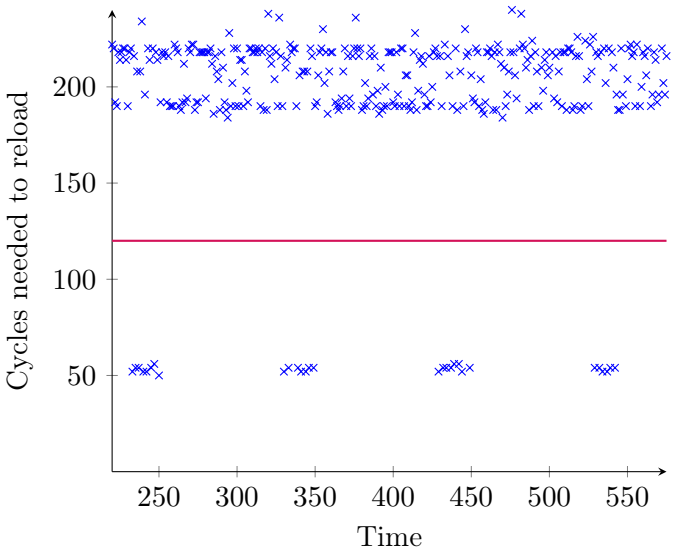


Figure 6.2. – Partial execution trace using the password `dAdinretAm`, username `admin` and salt `0102030405060708`.
Any value below the threshold represents an execution of the monitored instruction (*i.e.,* a different bit being processed). We can identify four groups, with the horizontal gap representing an overflow of the accumulator.

caused by the spy missing an iteration). In practice, it means that a three-bit pattern is considered as a four-bit pattern. Likewise, any isolated iteration (one or two-bit pattern) is removed from the trace since it can be considered an error in the measurement, or the consequence of a misprediction caused by the processor.

Then, we convert the trace into a readable form by only noting the number of iterations for each chunk in the trace. We used this representation as an input to our dictionary reduction program.

Dictionary reduction. This part of the attack is generic since it behaves as a classical dictionary partitioning attack, as described in chapter 4. The idea is to take a list of potential passwords and see if each candidate matches the trace we collected on the secret password. If the candidate does not match, we discard the password. Otherwise, we keep it and can apply the same process on a new leak (*e.g.,* changing the salt) to further reduce the list of candidates until an online attack becomes reasonable.

Because precise microarchitectural measurements are hard to get perfect, some minor errors may slip into our traces. Hence, in a realistic scenario, we cannot expect a perfect match between the trace we collect and the real trace of the password. To circumvent this constraint, we build a scoring system by computing a differential score for each candidate based on how different it is from the reference trace as follows.

Let x be the secret password, pwd be a password candidate and T_x be the trace array used as input (*i.e.,* T_w contains the length of the chunks we recovered).

1. We compute $x_{pwd} = H(salt \,||\, H(id \,||\, ":" \,||\, pwd))$, and simulate the exponentiation to get an array containing the length of each pattern in x_{pwd}: T_{pwd}.

2. Having T_{pwd} and T_x, we initialize the score s with the difference in the total number of observed iterations

$$s = \left| \sum_{i \in T_{pwd}} i - \sum_{j \in T_x} j \right|$$

3. For each pattern, starting from the last one, we add to the score the difference between the measured pattern's length, and the candidate's.

$$s = s + \sum_{i=1}^{len(T_x)} |T_x[-i] - T_{pwd}[-i]|$$

Thus, any minor issue during the measurement (*e.g.,* missing one iteration) has little impact on the overall score since the comparison is re-synchronized at each pattern. Furthermore, we start at the end to avoid the lack of precision of the first traces so that a potential missing part at the beginning does not de-synchronize the entire trace comparison.

For each candidate yielding a score below a predefined threshold (set to 10 in our experiments), the password and its score are returned as a valid candidate. If no candidate is returned (meaning there is an issue with the trace), the lowest-score candidate is returned as the most probable password.

6.4.3. Results

We tested our PoC on 100 different passwords, drawn randomly from the Rockyou dictionary [Nik09]. Since the exponent generation behaves as a random oracle seeded by the password, the username, and the salt, we kept a constant value for the salt and the username. Moreover, the password length and format do not impact the exponent or the leaked information.

For each password, we repeated our attack 15 times, yielding a total of 1500 different traces. Then, we parsed each trace independently and tried to recover the password in the dictionary. We also computed the average number of traces needed to achieve a reliable result by obtaining multiple traces on the same password.

Reliability of our measurements. In 96.9% of our experiments, a single measurement is enough to find the password with high probability. The result is clearly inconclusive on the remaining 3.1%, namely the errors, since the minimal score is above 20, whereas we often get a score below 10 when the valid password is found. Repeating the measurement twice was enough to achieve 100% accuracy.

It is worth noting that some measurements were completely inaccurate for unknown reasons, resulting in a non-exploitable trace. In this case, we performed the measurement again. This only represents a negligible amount of our measurements, and only occurred during intensive testing sessions.

Average information per trace. Due to the quick iterations prior to the use of the Montgomery representation (*i.e.,* before the first overflow of the accumulator), the beginning of the trace tends to be noisy. Thus, we could not reliably and consistently guess how many

MSB were skipped. Instead, we considered the total number of detected patterns, otherwise stated, the number of overflows on the accumulator during the exponentiation.

Our experiments show that we can guess 63.9 bits per trace on average. However, not all these guessed bits are accurate. Indeed, because of the noisy measurements, we notice an average difference of 3.9 bits with the reference hash value computed from the secret password. Nevertheless, this slight difference did not prevent us from correctly recovering the password (thanks to our scoring technique).

It is interesting to see that, in theory, given a 160-bit exponent, we expect an average leakage of 65.2 bits of information on x (see section 6.3 for details). Our practical results show that we can extract almost as much information as we expect from a single measurement. The slight loss of information, comes from the lack of precision at the beginning of the trace.

We stress that using a stronger hash function (*e.g.,* SHA-256 or SHA-512) would result in a longer digest, allowing an attacker to acquire even more information on the exponent and the password.

Practical dictionary attack. As discussed in subsection 6.2.3, recovering k bits of information on the exponent allows an attacker to lower the probability of having a false negative to 2^{-k} while eliminating wrong passwords.

The bottleneck of the attack being the dictionary reduction, we estimated its complexity in practice. As OpenSSL relies on SHA-1 in SRP, the cost of testing each password is dominated by the complexity of computing two SHA-1 digests on small data. For the following computation, we will assume that our attacker uses a single, publicly available, Amazon AWS p4d.24xlarge instance that runs eight NVIDIA A100 GPUs. Based on the Hashcat benchmark for SHA-1 [Tom21], a single GPU can compute $2.2 \cdot 10^{10}$ password candidates per second. Therefore, a single AWS instance can roughly evaluate $8.8 \cdot 10^{10}$ password candidates per second.

Table 6.1. – Cost of dictionary reduction for various dictionaries

Dictionary	Nb. entries	Reduction time	Cost ($)
Rockyou	$1.4 \cdot 10^7$	0.00016 s	$1.4 \cdot 10^{-6}$
CrackStation	$3.5 \cdot 10^7$	0.0004 s	$3.6 \cdot 10^{-6}$
HaveIBeenPwned	$5.5 \cdot 10^8$	0.00625 s	$2 \cdot 10^{-5}$
8 characters	$4.6 \cdot 10^{14}$	1h27min	47.58

Table 6.1 presents the cost for reducing some relevant dictionaries. We used the current price of an Amazon AWS p4d.24xlarge instance as a reference [Ama20], which is currently 32.77$ per hour. The dictionaries we present have been used in previous work [VR20] to demonstrate how practical offline dictionary attacks are on PAKEs.

Considering the size of these dictionaries and our experimental results, a single (reliable) measurement allows an attacker to recover enough information on the password to reduce the entire dictionary.

Comparison to other work. In parallel to our work, Russon [Rus21] unveiled a side-channel attack on Apple's CoreCrypto implementation of SRP. This attack shares some

similarities with ours: they also recover information during the computation of the verifier on the client side, and use it to perform a dictionary attack.

They highlight a leakage in the euclidian splitting protection used to mask the modular exponentiation internal computation (the bit length of the quotient may leak). This mild leak becomes significant if the attacker can repeat the measurements with the same exponent, which is the case for SRP. Our contribution differs in various aspects: (i) our attacker model is less intrusive since power analysis attacks usually assume physical access to the device; (ii) we attack a different exponentiation algorithm and acquire more information from a single measurement, allowing a single-measurement attack; (iii) our impact analysis suggests that our vulnerability has a broader impact, since it affects not only softwares using OpenSSL directly, but also the basic packages of various programming languages, widening the scope of impacted projects (see section 6.5).

> **Reminder:** We use the signal-to-noise metric to compare the side-channel efficiency (section 4.2). That is the ratio between the leakage for a fixed set of parameters and the number of measurements necessary for this leakage to be reliable.

Regarding the overall attack efficiency, for SHA-1, we get a signal-to-noise ratio of $\mathcal{D} = 63.9$ since a single measurement is enough. In comparison, the attack of Russon leaks 0.613 bits on average from 1000 measurements, yielding $\mathcal{D} = 6.13 \cdot 10^{-4}$.

6.5. Practical Impact

OpenSSL is one of the most popular and deployed cryptographic libraries [NKS+17a]. This explains why their SRP implementation is used as a reference for many amateur and industrial third-party projects. More interestingly, other popular languages, such as JavaScript, Ruby, and Erlang, rely on OpenSSL for big numbers operations. To our surprise, we discovered that the scope of our vulnerability is much broader than the OpenSSL TLS-SRP.

Since our attack targets the dynamic library, our exploit works on other vulnerable codes, regardless of the used programming language, as long as it is linked to the vulnerable version of OpenSSL. Due to the wide range of impacted projects, we did not implement each attack scenario but designed an exploit for OpenSSL, node-bignum, Apple HomeKit ADK, and PySRP and only evaluated the impact for other projects.

In this section, we go through the most popular open-source and proprietary projects we found vulnerable. In most cases, we noticed that the big numbers or the cryptographic libraries of a specific language are based on OpenSSL, making any SRP project based on this standard library vulnerable. Due to the wide variety of projects and the low code transparency of closed-source solutions, the actual scope of our vulnerability is hard to establish. Thus, we underline that this impact analysis is far from being exhaustive. Furthermore, practical attack consequences may vary between applications depending on the context and thus can hardly be narrowly defined. A general note on password recovery consequences in SRP can be found in subsection 6.2.4. Table 6.2 sums up the vulnerable projects that we identified in the wild.

Table 6.2. – Non-exhaustive list of vulnerable projects. Projects with a * are not patched. Vulnerable versions are inclusive.

Software/Library	Language	Vulnerable versions
OpenSSL	C	Up to v1.1.1i
Apple HomeKit ADK	C	no release
CMaNGOS[3]	C	no release*
GoToAssist	closed source	Up to v10.15.0*
node-bignum	JavaScript	Up to 0.13.1*
node-srp	JavaScript	Up to 0.2.0*
node-srp-typescript	JavaScript	Up to 0.3.0*
caf_srp	JavaScript	no release
OTP	Erlang	Up to 23.2.1
strap	Elixir	no release
srp-elixir	Elixir	0.2.0
ruby/openssl	Ruby	Up to 2.2.0
sirp	Ruby	Up to 2.0.0
pysrp	Python	Up to 1.0.16
xbee-python	Python	Up to 1.3.0*
proton-python-client	Python	no release

6.5.1. TLS-SRP using OpenSSL

As release 1.0.1, OpenSSL offers an implementation for TLS-SRP [TWMP07] that uses a shared password to establish a secure session. There are several reasons to use TLS-SRP. First, using password-based authentication does not require reliance on certificate authorities. Second, it provides mutual authentication, while TLS with server certificates only authenticates the server to the client. Third, phishing attacks are harder to perform since the authentication does not involve checking the URL being certified.

Our main contribution is illustrated by attacking this SRP implementation. Thus, one might argue that the scope of our work is limited for two reasons. Firstly, TLS-SRP is not supported by any browser and therefore is not widely used over the Internet. Secondly, TLS 1.3 does not provide any support for PAKE yet [BF18] and might never support SRP, since it is not part of the CFRG PAKE selection [CFR20].

Despite these reasons, we argue that the interest of our work is much broader than just TLS-SRP. Indeed, we found that the OpenSSL implementation is used as is in many other projects. In particular, the vulnerability is caused by an insecure call to the function `BN_mod_exp`, without setting the constant-time flag. Developers are not to blame here because the documentation of `BN_mod_exp` never mentions that, by default, this function is not constant time if the base fits in a processor word. Unfortunately, this is the case in SRP and might also be in other protocols.

[3]Contains multiple vulnerabilities and needs a complete review of their security.

6.5.2. Stanford Reference Implementation

Stanford University has dedicated a web page for SRP since the project started at it. The website hosts some pages describing the protocol evolution and its various use-cases. Namely, it provides reference implementations: a standalone Java implementation (relying on the default JDK) and a C implementation (relying on a third-party library to handle big numbers and cryptographic operations). We have studied both.

Java implementation. The Java implementation is utterly flawed and outdated, so we strongly discourage its usage. First, it is worth noting that the implementation is obsolete: some mitigations for existing attacks are missing, indicating we are not dealing with SRP-6a but a previous version. Here is a quick overview of the major flaws in Stanford's Java reference implementation of SRP:

- The API is basic and has overall poor security.
- It uses the generic `java.math.BigInteger` to handle big numbers (modular exponentiation, ...).
- It is not up to date:
  - The server sends $B = v + g^b \bmod p$. Since SRP-6a, the server sends $B = kv + g^b \bmod p$ to prevent an attacker from checking two passwords in an online dictionary attack.
  - Instead of computing $u = H(A \mid\mid B)$, it takes u as the 32 firsts MSB of $H(B)$. Hence, an attacker recovering v can impersonate any client.
  - There is no verification of either A or B, which enables the following attacks:
    - If $A = 0$, the server ends up with a predictable key $K = H(0)$.
    - Fixing $B = 0$ enables offline dictionary attacks.
- There is no group element verification, enabling small subgroup attacks, similar to the one described in Appendix A.
- The exponents used to compute the Diffie-Hellman key shares are only 64-bit long (the length is hardcoded). This enables an efficient dictionary attack:
  1. Given $A = g^a \bmod p$ (transmitted in clear), an attacker can recover a in $\mathcal{O}(2^{32})$.
  2. Knowing a, the only unknown in the client side computation of S is x, which is directly related to the password. Hence, the attacker may be able to perform an offline dictionary attack to recover the password by trying to compute S with a password candidate and verifying the confirmation message sent by the client. Since the computation for a single password candidate is $\mathcal{O}(3 * \texttt{SHA1} + 2 * \texttt{mod_exp})$, the overall attack complexity is $\mathcal{O}(2^{32} + n \times \mathcal{O}(3 * \texttt{SHA1} + 2 * \texttt{mod_exp})) = \mathcal{O}(2^{32})$ with n the number of password in the dictionary.

This implementation should be either re-implemented altogether or deleted. Nonetheless, it should not be presented as a reference since it does not include good security practices, is severely outdated, and presents very weak parameters by default.

C implementation. As for the C implementation, it only performs SRP-related operations: manages messages and parameters, keeps track of the protocol state, and calls the appropriate callbacks. A third-party library provides the underlying mathematical operations. The project supports and recommends the following libraries: OpenSSL, Cryptolib, GNU_MP, MPI, and TomMath/TomCrypt. OpenSSL is likely to be the choice of many developers among these libraries due to its large support in many systems. However, our study shows that the modular exponentiation API for OpenSSL uses the vulnerable function call of `BN_mod_exp`, making this default reference implementation vulnerable to our attack. Moreover, we verified that the generic attacks discussed in subsection 3.2.2 are possible since both the salt and the group parameters are transmitted without protection.

The Stanford implementation is noteworthy because it is one of the first available C implementations with no restrictive license. Therefore, it is fair to assume that it is used elsewhere. Nevertheless, it is not clear how many projects are impacted, especially the proprietary ones whose code is not open-source.

6.5.3. Apple HomeKit Accessory Development Kit

Apple HomeKit allows users to communicate with and control accessories (going from garage doors to smart light-bulb or doorbells) through a dedicated application. This communication goes through the HomeKit Accessory Protocol (HAP), deployed on over a billion Apple devices.

When a new device is first introduced into the network, the user must pair it with the application. To do so, a secure session is established with a modern variant of SRP. Namely, the following changes are made (Stanford's website is explicitly cited): (i) SHA-1 is replaced by SHA-512, and (ii) only the 3072-bit modulus group from [TWMP07] is supported. In this setup, the application acts as a client, the device is the server holding the verifier, and the password is a code displayed on the device. The SRP session is used during pairing to exchange long-term Ed25519 keys that are later used for all future sessions.

Apple's specification mandates a device code conforming to `XXX-XX-XXX`, meaning only 10^8 passwords are possible. Considering such a restricted set of password candidates and the reliability of our measurement, an attacker would be able to guess the code on-the-fly from a single session and recover the long-term key.

While commercial accessories must use Apple's HomeKit ADK through the MFi program, the open-source version relying on OpenSSL is available on Github [App20]. Unfortunately, this implementation calls `BN_mod_exp` on fresh big numbers (without modifying any flag) on the client device. Hence, it is vulnerable to our attack. The large exponent size, because of SHA-512, allows the attacker to get about 205.7 bits of information, which are more than needed to recover the device code. In this context, the consequences of our attack are disastrous: an attacker would have complete control of the targeted smart device if they could impersonate the user during the pairing.

As a side note, we stress that our current attack focuses on particular properties found on Intel processors, which are usually not valid for ARM processors (commonly used on iPhones, iPads, and recent Mac). Namely, the non-inclusiveness of the Last Level Cache (LLC) and the absence of a user-land flush instruction make it harder to perform cross-core attacks. However, all these obstacles can be circumvented, as demonstrated in the work of Lipp *et al.* [LGS+16]. We encourage readers to refer to this article for further details on how to adapt the attack for the ARM architecture. Furthermore, Mac computers manufactured before late

2020 integrate an Intel processor, and the Home application can be installed on them, making the current attack valid on many machines.

6.5.4. LogMeIn Tools

LogMeIn provides various tools for remote communications and collaboration services. We analyzed four of these tools: GoToMeeting (designed for visio-conferences), GoToWebinar (designed for webinars), GoToTraining (designed for teaching classes), and GoToAssist (remote IT assistance). According to the documentation we could find [Log20; Log17], all these tools have a common point: they use OpenSSL for their cryptographic implementation and rely on SRP for user authentication.

However, GoToMeeting, GoToWebinar, and GoToTraining seem to use SRP only for the web authentication part, and we could not confirm whether the web authentication is vulnerable to our attack. Our shallow reverse engineering analysis of the binaries finds that OpenSSL's SRP context is instantiated during a session, but we did not manage to identify its use.

As for GoToAssist, a whitepaper from 2017 [Log17] suggests that SRP is used to establish a secure session protecting screen-sharing, keyboard/mouse control, diagnostic data, and text chat. Thus, an attacker recovering the secret password may be able to put themselves in an active MitM position, achieving complete control over the victim's computer by impersonating the IT support until the end of the connection. Note that this analysis is only based on their available documentation, and that we did not perform the needed reverse engineering to verify their claims. Furthermore, LogMe recently rebranded as GoTo and updated its software and documentation in the process.

6.5.5. ProtonMail Python Client

According to the security description of ProtonMail [AG16], each user defines two passwords: the mailbox password used to protect the private key encrypting messages and the login password that is used along with SRP to provide remote authentication.

ProtonMail customizes its SRP implementation with several technical choices. First, they replaced SHA-1 with a combination of bcrypt and expanded SHA-512, so the attacker would require more time to compute the digest for each candidate (an attacker could test roughly $4.2 \cdot 10^5$ passwords per second, on the setup described in section 6.4). Note that using such a long exponent (2048 bits exponent) would leak a tremendous amount of information through our attack. Second, they use non-standard group parameters. Indeed, in order to avoid large-scale pre-computation attacks, they generate a new 2048-bit modulus for each user. However, since the generator is fixed to $g = 2$, our attack still applies. Moreover, opting for a random generator requires having a flexible API, enabling custom parameters. This choice, combined with unfortunately vulnerable dependencies, leads to another vulnerability that we detail in Appendix A.

Note that recovering the login password would allow an attacker to connect to the victim's account but not have access to their messages or their private key (assuming a different password has been used). However, the login password is shared with ProtonVPN. Therefore, recovering it allows the attacker to take complete control of the victim's ProtonVPN account, thereby benefiting from their subscription and viewing sensitive data, such as billing information.

It is worth noting that not all the implementations of ProtonMail are concerned by our attack. For instance, we did not find any OpenSSL call in the JavaScript client. In fact, only the python client is vulnerable, as it is based on the project pysrp. More details about this project are found in subsection 6.5.6.

6.5.6. Big Numbers Libraries and SRP in the Wild

To handle big integers, high-level programming languages typically have two choices: (i) re-implement a library or a package to give support; (ii) use an implementation that is provided by a low-level language. In practice, the latter is quite a popular approach, especially when considering efficiency during cryptographic operations.

A preliminary review of core libraries of various open-source languages reveals that OpenSSL, through different wrappers, is often used to manipulate big numbers. Therefore, these projects inherit the OpenSSL vulnerability. In particular, we observed that all SRP projects based on OpenSSL, or using a vulnerable SRP package, can be exploited by our attack. Namely, we identified the vulnerability in a JavaScript node to handle big numbers (node-bignum, used in more than 2500 other projects that we did not analyze), Ruby's official OpenSSL package, Erlang and Elixir OTP library, and a widespread Python SRP implementation.

We give details about the main packages we found to be vulnerable in the following subsections. The affected projects are being used as dependencies in numerous open-source and closed-source projects, so it is hard to assess the scope of their vulnerability.

JavaScript's node-bignum. Among the nodes supporting big numbers, some provide a full big integers implementation in pure JavaScript (bignumber.js or jsbn.js), while others rely on native libraries such as GMP (node-bigint) or OpenSSL (node-bignum). We only assess the security of node-bignum.

Although the project node-bignum may not be the most popular one to manipulate big integers in JavaScript (compared to bignumber.js, for instance), we found notable SRP projects relying on it. For instance, both Mozilla's node-srp and a more active fork node-srp-typescript leverage node-bignum to provide cryptographic operations on big integers. Thus, these two reference projects are directly vulnerable to our attack.

Ruby OpenSSL library. Reviewing the official Github repository of Ruby, we identified that it provides wrappers for some of the OpenSSL API, including the big integers operations. We found that the sirp project uses these wrappers in such a way that makes them vulnerable to our attack. In particular, Ruby did not provide any wrapper for the OpenSSL function that sets the flags to be constant-time. Therefore, no SRP project written in Ruby could be plausibly assumed to be secure, provided the reference OpenSSL package is used.

Erlang and Elixir. Erlang is mainly built upon the Open Telecom Platform (OTP), an extensive collection of general purposes libraries for Erlang. All the cryptography-related functionalities ensured by OTP are implemented through callbacks to OpenSSL. Hence, any projects based on OTP were vulnerable to our attack. Namely, OTP provides support for TLS-SRP, with the same implementation (and the same vulnerability) as OpenSSL. Given the list of the main companies using Erlang (*e.g.,* Whatsapp) [Fal], the range of the vulnerability may be considerable.

Elixir is built on top of Erlang, and, in particular, its cryptographic operations are provided by Erlang's OTP. Thus, some Elixir projects providing SRP implementations, such as strap and srp-elixir, were vulnerable to our attack.

Python SRP package. Although Python offers its own cryptographic packages, some projects choose to call OpenSSL directly in order to provide support for big numbers. More notably, the pysrp project implements SRP by leveraging OpenSSL, especially for the big integers operations. This package being the default Python implementation for SRP, it is used in at least ten other open-source projects[3], accumulating hundreds of stars on Github.

One project of particular interest is xbee-python (the official python library designed to interact with Digi XBee's radio frequency modules), which requires the vulnerable python package to authenticate with XBee devices over Bluetooth Low Energy. These radio devices can be deployed in a wide range of products for a broader range of applications (from intelligent lighting controls to storage tank monitoring and orbital experiments by the NASA [Fal17]).

6.6. Mitigations

Our attacks were performed on the most updated versions of the evaluated projects, as published at the time of discovery. Following responsible disclosure, we timely disclose our findings to all projects mentioned in Table 6.2. We further participated in the design and the empirical verification of the proposed countermeasures. We describe a brief overview of our communications.

The Case of OpenSSL. We started by contacting OpenSSL following their security policy. They acknowledged the attack and were able to reproduce it using our docker container. Therefore, they decided to mitigate the attack and include the patch in their next release[4]. To our surprise, they refused to publish a related CVE because they claimed that cache attacks are out of the scope of their threat model. We argued that a CVE might help to shed some light on the problem, as we cannot reach all the proprietary solutions relying on OpenSSL. However, the final decision was not to issue any CVE.

Then, we discussed with OpenSSL the best fix to deploy. Everyone agreed that the mitigation should be to prevent the function `BN_mod_exp` from calling the vulnerable function `BN_mod_exp_mont_word`. There are two options to apply this. First, the function `BN_mod_exp` is re-written so that it never calls the vulnerable function. Second, the call of the function `BN_mod_exp` is modified, so the flag `BN_FLG_CONSTTIME` is set on the exponent.

No option was perfect. Indeed, the first option would have increased the execution time of other code using the optimized modular exponentiation. However, it might fix all projects implementing SRP through the big integers API of OpenSSL. The latter option is easier to test and validate, as it solely concerns the SRP implementation in crypto/srp/srp_lib.c. However, it only fixes the particular SRP implementation of OpenSSL, not the projects directly calling `BN_mod_exp`. OpenSSL maintainers took a variant of the second option, as they also set the constant-time flag on the result of the function `BN_mod_exp`. Thus, we had no choice but to contact the vulnerable implementations that we previously identified.

[3] https://libraries.io/pypi/srp
[4] Release notes 1.1.1i to 1.1.1j

It is worth noting that the patch introduced some overhead since the optimized function is not called anymore. We empirically evaluated the incurred overhead by running our benchmark 30000 times with random values. Our results show an overhead of 11.18% on the execution of `SRP_Calc_client_key`. This overhead might explain the rationale for keeping the function `BN_mod_exp_mont_word` that might be used in applications requiring fast execution.

The Remaining Projects. We contacted the remaining projects since the OpenSSL patch does not cover them. Two projects are distinguished: Ruby and Erlang/OTP. Indeed, they quickly patched their implementation after our report and our validation of their suggested fix. Ruby ranked our vulnerability 8.7/10 in terms of severity and took care of notifying all Ruby-based projects. We designed and integrated a patch for four other projects: ProtonMail (who acknowledge our vulnerability through their bug bounty program), the JavaScript node-bignum, the Python pysrp, and the Apple Homekit. The patch for node-bignum is still to be integrated. Finally, we mention the cafjs, whose maintainer recognized the vulnerability but never replied to our subsequent emails suggesting a mitigation.

Remanent Side Channels in Dragonfly Implementations 7

In this chapter, we study implementations of Dragonfly from different projects (widespread Wi-Fi daemons and FreeRADIUS). We detail how we discovered and exploited three distinct side channels on the same protocol, two years apart. All of them lead to password recovery through a partitioning dictionary attack. In section 7.1, we set the bases of our contributions. We give motivations to study Dragonfly, set the threat model we consider, and give an intuitive attack scenario, that we use for all attacks. Then, in section 7.2, we present the first vulnerability we exploited in the iNet Wireless Daemon (iwd) and FreeRADIUS implementations. In section 7.3, we present a novel attack affecting hostap, iwd and FreeRADIUS, which escaped all previous analysis. We explain how this attack illustrates an untapped source of leakage that may still plague numerous implementations. In section 7.4, we compare our dictionary attacks to other work, both in terms of concept and efficiency. In section 7.5, we discuss the evolution of Dragonfly and the limitation of current mitigation strategies.

Takeaway:

This contribution supports the following conclusions:

- Recent Password-Authenticated Key Exchange protocols (PAKE) implementations are still prone to side-channel attacks.
- Leakage vectors may hide in third-party dependencies. Verifying the first layer of implementation is not enough to guarantee security.
- The implementation of a method that is constant-time by design (*e.g.,* SSWU) may not be secret independent.
- There is a need for a more sustainable mitigation strategy.
- Implementation concerns (*e.g.,* side-channel vulnerabilities) should be addressed during the standardization process.

The appropriate background to grasp the details of our contributions is provided in:

- Chapter 2 for the background to understand the microarchitectural attack. Section 2.3 provides a detailed description of Flush+Reload.
- Chapter 3, especially section 3.3, which gives an overview of the Dragonfly protocol, and its integration in Wi-Fi Protected Access (WPA) and EAP-pwd.
- Chapter 4 for a description of the generic partitioning dictionary attack, that we apply once we get the information on a password-related value.

Contents

7.1. Context and Motivation **79**
7.2. Dragonblood Is Still Leaking **81**
7.2.1. Attack Details 81
7.2.2. Vulnerable Implementations 82
7.2.3. Evaluation 86
7.2.4. Mitigations 90
7.3. Dragondoom: residual vulnerabilities in widespread implementations **90**
7.3.1. Attack details 91
7.3.2. Vulnerable Implementations 94
7.3.3. Evaluation 98
7.3.4. Impact on SAE-PT 102
7.3.5. Mitigations 102
7.4. Comparative Analysis **103**
7.4.1. Concept of the Attacks 103
7.4.2. Performance and Efficiency 103
7.5. Discussion **104**

7.1. Context and Motivation

> **Reminder:** Password-Authenticated Key Exchange protocols (PAKE) aim at establishing a secure and authenticated channel based on the shared knowledge of a password (chapter 3). Dragonfly is deployed as Simultaneous Authentication of Equals (SAE) in the newly standardized WPA3 (section 3.3).

Nowadays, there are more active Wi-Fi devices around the world than there are human beings. The Wi-Fi Alliance estimated Wi-Fi global economy value to be $3.3 trillion in 2021 and forecast its growth to $4.9 trillion by 2025 [All21]. Such ubiquity and economic worth make protecting Wi-Fi vital. Since 2003, the Wi-Fi Alliance has introduced three major versions of its security protocol: the Wi-Fi Protected Access (WPA).

The most recent one dates back to 2018, when WPA3 was announced in response to several serious weaknesses being identified in WPA2 [VP16; VP17; VP18]. A distinctive security feature of WPA3 is to leverage a PAKE, called Dragonfly, to protect users' passwords from the offline dictionary attacks that haunted the WPA2 handshake authentication. Despite being recently released, WPA3 already knows large adoption from major Wi-Fi providers and software; in particular, it has become mandatory for Wi-Fi certification since July 2020 [All20a].

The rising popularity of Dragonfly, or its WPA3 variant called SAE, and the controversy raised by several Crypto Forum Research Group (CFRG) members [Per13; Flu14], motivated research works to assess the security of its deployed implementations. Notably, Vanhoef and Ronen [VR20] presented the Dragonblood attacks, one of which exploits a microarchitectural side-channel in multiple WPA3 implementations to recover users' passwords.

The disclosure of Dragonblood is unfortunate to the Wi-Fi Alliance that has just got its biggest update in 14 years. In response, they published implementation guidance to be followed by manufacturers [All19] to ensure secure backward compatible WPA3's implementations. Authors in [VR20] cast doubts on the endorsement of some backward-compatible side-channel defenses, especially in resource-constrained devices because of their high overhead. Moreover, they argue that a secure implementation of the countermeasures is an arduous task.

Interestingly, Dragonfly is also deployed in FreeRADIUS, in the Extensible Authentication Protocol (EAP) authentication scheme when using pwd mode. While FreeRADIUS claims to be deployed on millions of devices[1], the particular mode is not the most popular, and it is hard to assess its practical usage. Hence, our main scenario focuses on the Wi-Fi daemons. Still, the underlying technical details are similar concerning the identified vulnerability in the Wi-Fi daemons and FreeRADIUS, and we will provide insight into the impact of our attacks on both contexts.

In this chapter, we study the Dragonfly implementation of multiple projects: hostap, iwd, and FreeRADIUS. First, we look closely at their implementation of the suggested mitigations (section 7.2). Then, we study the interaction between the daemons and their third-party dependencies in charge of performing sensitive operations: the cryptographic library (section 7.3).

[1] https://freeradius.org/about/#usage_statistics

> **Reminder:** As a default model, we consider the classical FLUSH+RELOAD model (section 5.2). It assumes an unprivileged spyware running on the victim's computer.

Threat Model. As is often the case when targeting Wi-Fi daemons, we consider an attacker with some physical proximity (*i.e.,* within network range). We also assume they can monitor multiple handshakes using the same password while varying at least one Media Access Control (MAC) address. This can be achieved differently depending on the target. First, if the target is an Access Point (AP), the attacker can try to connect, triggering an (invalid) handshake (the relevant part of the handshake will execute even if the password is invalid). Second, suppose the target is a client. In that case, the attacker can set up a fake AP to impersonate a valid one (known to the user) by spoofing the original AP and advertising the same Service Set Identifier (SSID) with a stronger signal strength (making it the default choice for the client Wi-Fi daemon). In both cases, the attacker can control the MAC address used by the targeted device. Setting a fake AP comes with other benefits. Indeed, they can force authentication using SAE, with the default password derivation, by simply omitting the corresponding Extended RSN Capabilities field.

> **Reminder:** When a password-related value is computed from an unprotected public value, an attacker may alter the public value and repeat their measurement to aggregate information on the password (subsection 4.1.1).
> In SAE, the password-related values are computed from the password and the MAC addresses of the client and AP (section 3.3).

Attack Scenario. We suppose a classical infrastructure where clients communicate with an AP across a wireless network. The attacker's goal is to steal the password used to establish a secure communication with the AP. Once the password is compromised, the attacker can enter the network and perform malicious activities.

In order to leverage the vulnerabilities defined in this chapter, the attacker must perform two tasks. First, they need to install an unprivileged spyware on a client station. Second, they need to create a rogue AP that behaves as the legitimate AP, but can use different MAC addresses for different connections.

Of course, we suppose that the rogue AP does not know the correct password, and therefore any session establishment between the rogue AP and a valid client will fail. Here, the rogue AP aims to state different MAC addresses and trick a client device into starting a Dragonfly key exchange. Thus, using the correct password, the Wi-Fi daemon will perform some operations that the attacker spy process will monitor. For each of these (failed) connections, the spyware will generate a new trace that leaks information on a password-related value, building a fingerprint of the password. This fingerprint is then used offline to prune a dictionary by verifying if the candidates match the fingerprint.

7.2. Dragonblood Is Still Leaking

> **Reminder:** In the early version of Dragonfly, the password is converted into an elliptic curve point using a (probabilistic) try-and-increment method called *hunting-and-pecking* (section 3.3). Knowing how many iterations it takes to convert a password is valuable information for an attacker.

In this section, we focus on the recommendations related to *Cache-Based Elliptic Curve Side-Channels* in [All19], which address mitigations to the set of Dragonblood vulnerabilities related to cache-based attacks in hunting-and-pecking [VR20]. Two mitigations are underlined: (i) performing extra dummy iterations on random data and (ii) blinding the calculation of the quadratic residue test. For the first mitigation, the RFC 7664 [Har15] recommends that 40 iterations are always executed even if the password was successfully encoded requiring fewer iterations. Concerning the second mitigation, a blinding scheme is suggested for the function that determines whether or not a value is a quadratic residue modulo a prime.

We show that such countermeasures are insufficient to defend against cache-based side-channel attacks. In fact, these particular measures are designed to prevent only *a part* of Dragonblood's attacks, and do not affect one of them. Especially, the cache attack, leveraging a password-dependent control-flow of the loop in the try-and-increment conversion function, is neither discussed in [All19] nor patched in most implementations (except for hostap/wpa_supplicant, which were the direct targets of the initial attack). Some projects seem aware of the issue and decided to overlook such an attack and prioritize patching other vulnerabilities [OSC19].

To demonstrate the reach of this vulnerability, we extend the original attack in which only the outcome of the first iteration is leaked. Using an unprivileged spyware, we demonstrate that an attacker can learn the exact iteration where the first successful conversion occurred with high probability, effectively doubling the leakage with half the measurements.

We only address the concern of the legacy SAE, as SAE-PT and SAE-PK were not part of the standard at the time of this contribution. In particular, we focus on the elliptic curve variant of SAE, as it is the only mandatory groups to support.

7.2.1. Attack Details

> **Takeaway:** In unpatched hunting-and-pecking, the control flow of the conversion loop varies when the password is successfully converted. An attacker may abuse this leakage to learn the exact iteration converting the password into a point, despite a constant number of iterations.

A naive implementation of hunting-and-pecking, complying with the Wi-Fi Alliance recommendations, should not leak information through the overall execution time. This is usually ensured through random masking and additional dummy operations. We provide an illustration of such implementation in the background section, in Listing 3.1, that we will reference in the current section.

Leakage Origin. Despite the overall constant execution time, the control flow of each iteration is *not* secret-independent. In fact, `x_cand` is computed from the secret password, the explicit branches at line 8 and line 9 leaks information on the password. Namely, the first

branch is taken whenever `x_cand` corresponds to the x-coordinate of a point on the curve, and the second branch is only taken once, on the first successful conversion. Moreover, upon conversion, the password is replaced by a random dummy value, and a boolean is set, so the second branch is not retaken. This makes any subsequent information independent from the password.

Hence, an attacker who can tell at what iteration the code between line 10 and line 12 is executed can guess how many rounds are needed before successfully returning from this function.

Leakage Impact. The amount of leaked information depends on the probability that an event happens. When converting a password into a group element, the success of each iteration is the success of the quadratic residue test over `x_cand`. Let p be the order of the underlying field and q the order of the generator. Since Dragonfly only supports elliptic curves of cofactor $h = 1$, q also denotes the total number of points on the curve. Then, a random integer `x_cand` $\in [0, p)$ is a quadratic residue with probability:

$$p_s = \frac{q}{2p} \approx 0.5.$$

Since this value is the output of a Key Derivation Function (KDF), we can consider that each `x_cand` are uniformly distributed in $[0, p)$. Hence, each iteration is independent of the others, and the probability for any password to be converted successfully at iteration n is $pr_n = p_s^n$.

This means that depending on the iteration that converts the password successfully, the leakage changes drastically: a conversion at the first iteration leaks 1 bit, while a conversion at the 10^{th} iteration would leak roughly 10 bits. Let k be the fixed number of iterations, as defined in the guidance of the RFC (a common assumption would be $k = 40$). For a given set of public MAC addresses (A, B), an attacker can expect an average leakage of

$$\ell_{A,B} = -\sum_{i=1}^{k} pr_i \cdot \log_2(pr_i) \approx \sum_{i=1}^{k} \frac{i}{2^i}.$$

When k grows, this sum quickly converges, giving an average leakage of 2 bits for a fixed set of public parameters. The attacker can aggregate leakage for multiple sets of public parameters, and perform a dictionary partitioning attack as described in chapter 4.

7.2.2. Vulnerable Implementations

> **Takeaway:** For hostap, the mitigations implemented by the authors of Dragonblood prevent our attack. On the other hand, both iwd and FreeRADIUS are vulnerable and leak 2 bits of information on average.

As the leakage comes from improper handling of the conversion iterations, the vulnerability of a project depends on its implementation. We study three main implementations, at their current version: hostap v2.9, iwd v1.8, and FreeRADIUS v3.0.Z21. We chose these projects because they were vulnerable to the initial Dragonblood attack, are supposed to be patched, and are open-source.

```
1  bool sae_compute_pwe(struct l_ecc_curve *curve, char *pwd, const uint8_t *a, const uint8_t *b) {
2      uint8_t seed[32], save[32], random[32], *base = pwd;
3      l_ecc_scalar *qr = sae_new_residue(curve, true);
4      l_ecc_scalar *qnr = sae_new_residue(curve, false);
5
6      for (int counter = 1; counter <= 20; counter++) {
7          /* pwd-seed = H(max(a, b) || min(a, b), base || counter)
8           * pwd-value = KDF(seed, "SAE Hunting and Pecking", p) */
9          sae_pwd_seed(a, b, base, base_len, counter, seed);
10         pwd_value = sae_pwd_value(curve, seed);
11         if (!pwd_value)
12             continue;
13
14         if (sae_is_quadradic_residue(curve, pwd_value, qr, qnr)) {
15             if (found == false) {
16                 l_ecc_scalar_get_data(pwd_value, x, sizeof(x));
17                 memcpy(save, seed, 32);
18                 l_getrandom(random, 32);
19                 base = random;
20                 base_len = 32;
21                 found = true;
22             }
23         }
24         l_ecc_scalar_free(pwd_value);
25     }
26     /* ... */
27 }
```

Listing 7.1 – *Hunting-and-pecking* on ECP group as implemented in iwd. Variable names have been adapted for a better fit.

Our study shows that following the disclosure of Dragonblood, the hostap provides secret-independent implementations of the loop. Therefore, it is not vulnerable to our attack and will not be further discussed. On the other hand, both other projects are vulnerable. Hereafter, we present the implementation of iwd and FreeRADIUS, and detail how they are affected. We also underline additional vulnerabilities that we discovered during our analysis.

iwd/ell

The wireless daemon iwd aims to replace wpa_supplicant (Wi-Fi client from hostap). The version 1.0 was released in October 2019 (after the publication of Dragonblood), and Arch Linux and Gentoo have already adopted it. Their implementation of Dragonfly follows the standard SAE [IEE16]. Only the ECP-groups variant is supported with the NIST's curves P256 and P384. The corresponding *hunting-and-pecking* is implemented in the function `sae_compute_pwe`, illustrated in Listing 7.1.

Each type or function starting by `l_*` refers to a function in the Embedded Linux Library (ell), a minimalist cryptographic library developed by Intel. By default, this library is statically linked to the binary at compilation time. However, users can decide to use a dynamic linking by specifying the correct option before compiling. We stress that the linking strategy does not impact the result of our attack; only some details in the addresses to monitor are concerned (see subsection 7.2.3).

A review of the code show that line 14 and line 15 result in branches corresponding to our vulnerability. An attacker who can learn when line 16 to line 21 are executed can learn two

bits of information on average, as described previously.

Miscellaneous leak. As specified in the Dragonfly RFC [Har15] and in the SAE standard [IEE16], the number of iterations to perform during the password conversion is not fixed. It can be defined as any non-negative integer, providing it guarantees a successful conversion with high probability. RFC 7664 advises setting k to at least 40 to get roughly one password over one trillion that needs more iterations. As for iwd, the implementation sets $k = 20$, making this probability significantly lower, with about one over $2 \cdot 10^6$ passwords requiring more than k iterations. In practice, using only passwords drawn from existing dictionaries [Nik09; Sec], we were able to find a consistent list of passwords needing more than 20 iterations. In this scenario, a client would be unable to authenticate to the AP until the password or the MAC address of one party is changed.

More importantly, the probability of needing more than 20 iterations is high enough for an attacker to generate such a scenario intentionally. For example, they can do so for a fixed password by repeatedly changing the MAC address of the rogue AP until the handshake fails due to an improper conversion. In such a case, the attacker gets roughly 20 bits of information on the password with a completely remote attack (no cache attack is involved in this scenario). However, they need to perform roughly 10^6 handshake, while changing the MAC address of the rogue AP.

FreeRADIUS

> **Reminder:** EAP-pwd differs from SAE in a few details (subsection 3.3.4): (i) the constant number of iterations in the conversion is not mandatory; (ii) the server embeds a random token in the password conversion; (iii) labels values are different.

FreeRADIUS supports EAP-pwd, a variant of Dragonfly, as a non-default authentication method, encapsulated in the RADIUS protocol. Besides the patches to Dragonblood attacks, we show that EAP-pwd is still vulnerable to timing attacks (due to a variable number of iterations) and to the same cache attack as iwd.

FreeRADIUS implementation. The Dragonfly exchange implemented by FreeRADIUS follows EAP-pwd's specification [ZH10]. All related functions are defined in the according module[2]. Namely, hunting-and-pecking is implemented in the function `compute_password_element`, as illustrated in Listing 7.2. We removed or reduced some parts of the code and renamed variables for clarity.

This implementation heavily relies on OpenSSL to perform cryptographic operations, such as hashing, manipulating big integers, and elliptic curve points. By default, the library is dynamically linked from the system-wide installation when building the project. A quick look at the code in Listing 7.2 shows a few branches inside the loop:

- At line 29, the iteration will end if the output of the KDF is bigger than the prime defining the underlying field. Considering the supported curves, this happens with a negligible probability on a random number of such size.

[2] https://github.com/FreeRADIUS/freeradius-server/tree/v3.0.x/src/modules/rlm_eap/types/rlm_eap_pwd

```
1  int compute_password_element (pwd_session_t *session, uint16_t grp_num, char const *pwd,
2              int pwd_len, char const *id_server, char const *id_peer, uint32_t *token) {
3      /* Instantiation of some variables and contexts... */
4      ctr = 0;
5      while (1) {
6          if (ctr > 100)
7              goto fail;
8          ctr++;
9
10         // pwd-seed = H(token | peer-id | server-id | pwd | ctr)
11         H_Init(ctx);
12         H_Update(ctx, (uint8_t *)token, sizeof(*token));
13         //  [...]
14         H_Final(ctx, pwe_digest);
15
16         // prfbuf = KDF(pwe_digest, "EAP-pwd Hunting And Pecking", p)
17         BN_bin2bn(pwe_digest, SHA256_DIGEST_LENGTH, rnd);
18         if (eap_pwd_kdf(pwe_digest, SHA256_DIGEST_LENGTH,
19                 "EAP-pwd Hunting And Pecking",
20                 strlen("EAP-pwd Hunting And Pecking"),
21                 prfbuf, primebitlen) != 0)
22             goto fail;
23         BN_bin2bn(prfbuf, primebytelen, x_candidate);
24
25         /* Handle big number conversion issue ... */
26         if (primebitlen % 8)
27             BN_rshift(x_candidate, x_candidate, (8 - (primebitlen % 8)));
28         if (BN_ucmp(x_candidate, session->prime) >= 0)
29             continue;
30
31         // need to unambiguously identify the solution, if there is one...
32         is_odd = BN_is_odd(rnd) ? 1 : 0;
33
34         /* solve the quadratic equation, if it's not solvable then we
35          * don't have a point */
36         if (!EC_POINT_set_compressed_coordinates_GFp(session->group,
37                 session->pwe, x_candidate, is_odd, NULL))
38             continue;
39
40         // Check if the point is on the curve
41         if (!EC_POINT_is_on_curve(session->group, session->pwe, NULL))
42             continue;
43
44         if (BN_cmp(cofactor, BN_value_one())) {
45             /* make sure the point is not in a small sub-group */
46             if (!EC_POINT_mul(session->group, session->pwe, NULL, session->pwe,
47                 cofactor, NULL))
48                 continue;
49             if (EC_POINT_is_at_infinity(session->group, session->pwe))
50                 continue;
51         }
52         /* if we got here then we have a new generator. */
53         break;
54     }
55     /* Clean allocated memory and handle errors ... */
56 }
```

Listing 7.2 – FreeRADIUS code sample for hunting-and-pecking, extracted from `eap_pwd.c`.

- At line 38, if the candidate is not an x-coordinate of a point on the curve, the rest of the loop is skipped. The same phenomenon occurs at line 42.

- Finally, at line 53, the loop ends if the password is converted.

The last item makes the total number of operations password-dependent. This issue is well known and was described in the original attack [VR20], and can lead to a remote timing attack.

Furthermore, the branch line 38 implies that the remaining code in an iteration would only be executed if the password is successfully converted. Since this code would only be executed once, this makes our attack possible.

7.2.3. Evaluation

> **Takeaway:** By monitoring two addresses (one as a synchronization clock, the other being the distinguishing operations) using Flush+Reload and repeatedly evicting a single memory address, we can reliably leak the number of iterations needed to convert a password.

We demonstrated our attack on iwd version 1.8 (current version as of the time of testing), and FreeRADIUS v3.0.21 but we believe that our work applies to any unpatched implementation that is still vulnerable to the initial cache-attack. All the scripts and programs we used to implement a Proof of Concept (PoC) attack against iwd and FreeRADIUS are made open-source[3, 4]. Since the attack applies similarly to both projects, we only detail the case of iwd. Unless stated otherwise, all line numbers in the following subsections refer to Listing 7.1.

Experimental Setup

Our experiments were performed on a Dell XPS13 7390 running on Fedora 31, kernel 5.6.15, with an Intel(R) Core(TM) i7-10510U and 16 GB of Random Access Memory (RAM). Binaries were compiled with gcc version 9.3.1 build 20200408 using the default configuration (optimization included). Namely, the Embedded Linux Library version 0.31 was statically linked to iwd during compilation.

During our experiment, we deployed hostapd (version 2.9) as an AP, and iwd (version 1.8) as a client. Both were installed and launched on the same physical device, using emulated network interfaces, as described in [Nik19].

We kept the default configuration on both ends, meaning the key exchange is always performed using IKE group 19, corresponding to P256. Similar results would have been observed using group 20 (curve P384) by tweaking the threshold of our spy process.

Our spy process has been implemented by following classical Flush+Reload methods. Moreover, we used Mastik v0.02 implementation inly for the Performance Degradation Attack (PDA) [Yar16].

[3] https://gitlab.inria.fr/ddealmei/poc-iwd-acsac2020
[4] https://gitlab.inria.fr/msabt/attack-poc-freeradius

Practical Attack Against iwd

In order to efficiently determine at which iteration a password is successfully converted, the attacker's needs are twofold. First, they need to be able to distinguish each iteration. Second, they shall guess when the success-specific code (between line 16 and line 21) is executed.

> **Reminder:** FLUSH+RELOAD enables the attacker to spy on the access to specific instructions if they are from shared memory (*e.g.,* shared library). A Performance Degradation Attack (PDA) is often used to slow the execution of other instructions (subsection 2.3.2).

Distinguishing iterations. To achieve the first goal, we create a synchronization clock by monitoring a memory line accessed at the beginning of each loop. The call to `kdf_sha256`, a function of `libell` called inside `sae_pwd_value`, is a good candidate. More specifically, we monitor a memory address corresponding to the loop calling this hash function. Thanks to this operation's complex nature, we could detect access to this call every time. Moreover, this operation is not used elsewhere in the protocol, thereby avoiding any potential noise in our traces.

Leaking successful conversion. Monitoring access to the code executed on success is less straightforward: the address range to be accessed inside `sae_compute_pwe` is too small and too close to the rest of the loop to be reliably monitored. The best choice is to monitor instructions in one of the functions called at line 16, line 17 or line 18. Tests have shown that monitoring inside `l_getrandom` yields the best results: other functions are called too often, at various places, bringing noise to our traces. However, random number generation is also part of the quadratic residue verification (`sae_is_quadratic_residue`, line 14) in order to blind the computation. Fortunately, these accesses can be distinguished given the number of cycles elapsed since the beginning of the iteration.

> **Reminder:** Testing the validity of an x-candidate implies a masked Legendre symbol computation (*i.e.,* modular exponentiation), as discussed in section 3.3.

Improving reliability. Due to complex CPU optimization techniques (see section 2.1) and some system activity, the measurements are noisy, and some traces may yield incorrect results. Moreover, a call to `l_getrandom` is usually performed in a few cycles, implying that we can miss it due to the temporal resolution of FLUSH+RELOAD. In order to significantly improve the reliability of our results, we combined the FLUSH+RELOAD attack with the PDA. Since the first call to `l_getrandom` occurs before the proper quadratic residue check, we evict a memory line inside the code in charge of the Legendre symbol computation. Hence, we significantly increase the delay between our synchronization clock and the success-specific code while keeping a low delay to reach the first call to `l_getrandom`.

Trace Collection

Once both client and AP are setup to use a password randomly drawn from a dictionary, we launch the spy process to monitor our probes. After each connection, we disconnect and reconnect the client to make multiple measurements. This step emulates a de-authentication

```
# First three lines correspond to the qr and qnr generation.
# They are ignored during parsing
l_getrandom 5435937
l_getrandom 5439791
l_getrandom 5455232
# First call to kdf_sha256 means the loop starts
kdf_sha256 5459308
kdf_sha256 3324
kdf_sha256 4091
kdf_sha256 3972
l_getrandom 108 # a low delay (108) indicates a call for masking
# At the fourth iteration, we notice a long-delayed call to l_getrandom.
# It means we can stop there.
l_getrandom 3889
kdf_sha256 417
l_getrandom 50
kdf_sha256 5691
l_getrandom 129
kdf_sha256 3795
# Other long-delayed calls can be observed,
# hence the need to acquire multiple samples
l_getrandom 4320
kdf_sha256 4524
...
```

Listing 7.3 – Truncated measurement yielding four iterations. This was acquired using the password `superpassword`, with MAC addresses `E2F754FE22D1` and `9203835A576B`. Annotations (#) have been added and are not part of the original trace.

attack aiming at collecting multiple samples with the same MAC addresses. For each password, we go through this process using ten different MAC addresses, allowing us to acquire up to 10 independent traces for the same password. For each MAC address, we make 15 measurements. Our observations were consistently obtained through testing 80 passwords to evaluate the effectiveness and the reliability of our trace collection techniques.

> **Reminder:** A *measurement*, or *sample*, is the result of monitoring one execution. A *trace* is a collection of measurements, with a fixed set of parameters, representing the leaked information.

We call *measurement*, or *sample*, the result of monitoring one Dragonfly key exchange with a fixed password and MAC addresses. It is the output of the spyware, represented by a succession of lines, corresponding to either a call to the synchronization clock (`kdf_sha256`) or `l_getrandom`. The value following each label indicates the delay since the last call to the synchronization clock. An example can be found in Listing 7.3, corresponding to a measurement yielding four iterations. A *trace* is a collection of measurements, all corresponding to the same password and the same MAC address.

Trace Interpretation

We designed a script that automatically interprets our traces and outputs the most probable iteration in which the process of password conversion first succeeds.

The core idea is to reduce the noise by eliminating all poorly formed measurements (which could not be interpreted, often because of system noise). Then, each measurement is processed independently, contributing to creating a global trace score. To do so, each line of a sample is read and processed according to its label. If the label is the synchronization clock, we increase the iteration counter by one. Otherwise, the score of the current iteration is increased by the delay associated with that line. In the latter case, if the delay is long enough (the threshold may be architecture specific), we can stop the parsing of that sample and process the next one. Once every sample of a trace has been processed, the score of each iteration comes as an indicator of the most probable iteration converting the password.

Since false positives have severe consequences, we chose to eliminate any trace that does not yield a precise result. In such a case, the script raises a warning to the attacker for future manual interpretation.

Performance and Accuracy

We summed up the results of our experimentations, with various numbers of samples for each MAC address, in Figure 7.1. With only one measurement per trace, approximately 70.5% of the traces can be automatically interpreted (others have a high risk of miss-prediction). However, the accuracy of our prediction is only 66%. We need to collect five samples to achieve an accuracy greater than 90% (with 77% of usable traces). We achieve 99% accuracy with only ten measurements, with a trace usability of 88%.

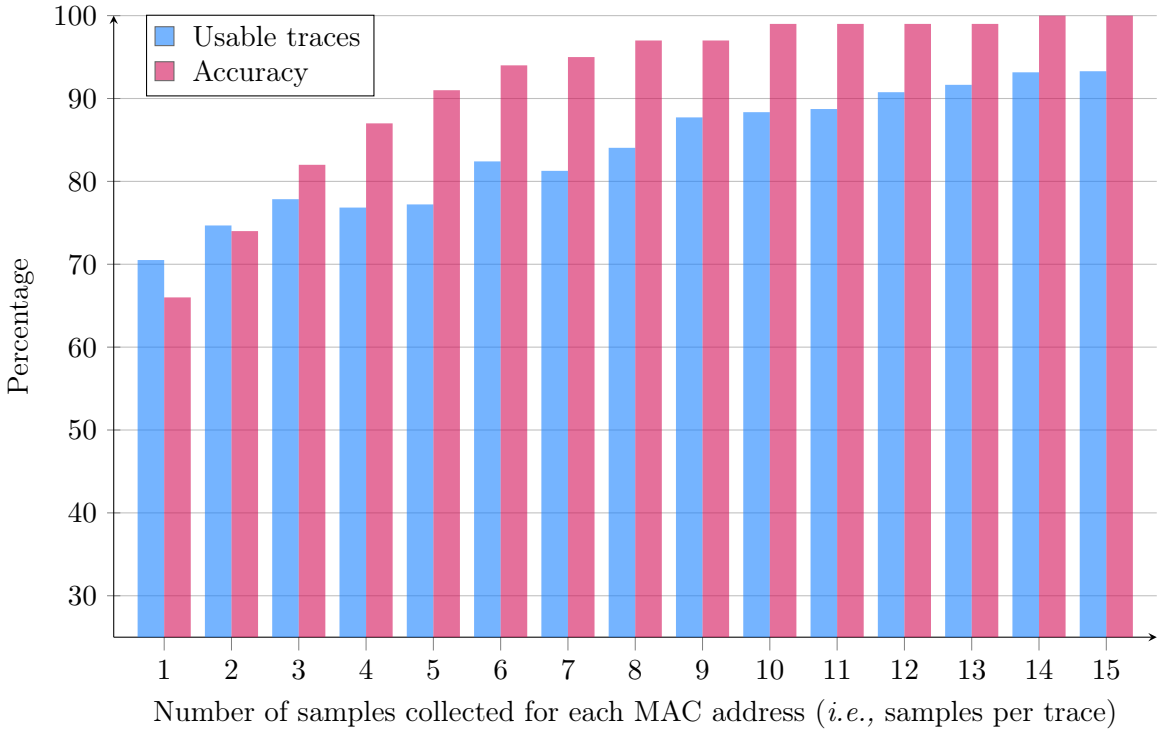


Figure 7.1. – Reliability of our experiment given a different number of samples to interpret for each MAC address. Accuracy represents the closeness of our prediction to the real value. Usable traces represent the percentage of traces we were able to *automatically* exploit, without high risk of miss-prediction.

We stress that trace usability only represents the ability of the *parser* to *automatically* interpret the trace. For most warnings, a manual reading of the samples (about 1-2 minutes) allows the attacker to successfully predict the round (some measurements do not yield a clear result and should be ignored). We also note that even if our script could not decide between two adjacent values (*e.g.,* five and six), we can assume that more than four iterations are required for password conversion.

A comparative analysis of this attack and the original Dragonblood is provided later, in section 7.4.

7.2.4. Mitigations

Following the disclosure of Dragonblood, several mitigations have been proposed [Har19a; Har19b] to replace the iterative hash-to-group function by a deterministic function. Since our contribution, this alternative method has been integrated into the standard, and the new password conversion function is used in SAE-PT and SAE-PK. However, backward compatibility might be a requirement in the industry, foreshadowing the availability of the legacy password conversion. Hence, we encourage a branch-free loop implementation to avoid any residual leakage.

We implemented such mitigations into iwd and FreeRADIUS, inspiring ourselves from hostapd patch[5]. We estimated the overhead induced by such countermeasure using the `rdtsc` assembly instruction, which offers high precision. We made 10,000 measurements for both the mitigated derivation and the original one while varying the password. We observed a negligible overhead ($1.4 \cdot 10^{-9}$% on average). The code complexity is barely affected by our changes. Considering the attack impact and the minor downside of the patch, we strongly recommend that developers include it in their products. Following our discoveries, both iwd and FreeRADIUS have smoothly integrated our patch into their code base.

Additionally, iwd switched from 20 to 30 iterations to lower the probability of a failure in the password conversion while keeping good performance.

We further discuss Dragonfly's evolution and this mitigation strategy in section 7.5.

7.3. Dragondoom: residual vulnerabilities in widespread implementations

> **Reminder:** Starting from 2020, WPA3 support both SAE and SAE-PT. The former uses the hunting-and-pecking password conversion (section 3.3). The latter switch to a deterministic conversion based on Simplified Shallue-van de Woestijne-Ulas (SSWU) [BCI+10; FSS+22].

After much effort, one might reasonably presume that the attack vectors as presented in [VR20] and section 7.2 ([BFS20a]) be no longer relevant. Namely, eminent open-source WPA3 implementations (*e.g.,* hostap) are expected to be exempt from microarchitectural side-channel attacks, especially since they were analyzed manually and by leveraging automatic tools to detect microarchitectural leaks, such as MicroWalk [WMES18a]. In this section, we challenge this belief and raise the question again of whether Dragonfly is secure in practice. Our approach consists of analyzing and tracking password-tainted values within

[5] https://w1.fi/security/2019-1/

the password conversion operation of Dragonfly, especially inside calls to external functions. Indeed, Dragonfly implementations rely primarily on third-party libraries to perform their required cryptographic operations. These libraries do not always provide "constant-time" (secret-independent) implementations for their functions. Unfortunately, the security impacts of calling an external leaky function within Dragonfly were left unstudied.

Our intuition comes from the fact that previous works only discovered leakages caused by the WPA3 code itself; in particular the password-conversion loop (line 4 to line 12 of Listing 3.1). However, calls to external functions were not studied. Some of the called functions are not secret-independent because they mostly deal with public values. Nevertheless, they do involve password-dependent values in Dragonfly.

7.3.1. Attack details

> **Takeaway:** Generic implementation of functions handling big numbers may not consider all inputs as secret. Hence, they may leak information on specific parameters if they are used in a specific context, where secrets are embedded in (usually public) inputs.
> This may be the case for SAE's elliptic curve point decompression method since the point format is related to the secret.
> Additionally, upon conversion of a buffer into its big number representation, Most Significant Bit (MSB) to zero may be skipped to avoid additional processing. This also leaks information.

We look carefully into the part of the code making external calls with secret-related values. Here, we did not apply sophisticated tools, demonstrating that existing implementations are far from secure. Indeed, we perform a taint analysis of the password conversion function using a basic but scalable tool (timecop [Nei]), combined with manual analysis. We highlight two main leakages. First, when the hunting-and-pecking is successful, the point decompression leaks a relation between a point coordinate and a password-related value (which affects SAE and EAP-pwd). Second, the binary-to-big-number routine called on various values, including secrets, leaks how many MSB of the input are zeros (which affects SAE, EAP-pwd, and SAE-PT).

Of particular interest, we study widely used libraries, namely OpenSSL, WolfSSL, and ell, supported by popular SAE (and EAP-pwd) implementations: hostap, FreeRADIUS, and iwd. Our findings reveal that all of these libraries are affected by at least one vulnerability. Consequently, projects relying on the generic point decompression function exposed by these cryptographic libraries leak secret information. One peculiar example is Apple CoreCrypto which provides its own SAE implementation, where it re-implements routines to be secret-independent. A deeper analysis of all these implementations and their differences is given in subsection 7.3.3.

Point Decompression

> **Reminder:** In the hunting-and-pecking password conversion, an x-coordinate corresponding to a point is found with overwhelming probability after the try-and-increment loop (section 3.3).

```
1  def set_compressed_coord(x, fmt, ec):
2      y_sqr = x^3 + ec.a·x + ec.b mod ec.p
3      y = sqrt_mod(y_sqr, ec.p)
4
5      if ( y mod 2 ≢ fmt mod 2 )
6          y = ec.p − y
7
8      P = (x, y)
9
10     return P
```

Listing 7.4 – Point decompression algorithm. `x` is the coordinate of the point, `fmt` represents its compression format (3 or 4), and `ec` is the structure of the elliptic curve.

Leakage Data: Seed Parity Since a single x-coordinate can describe two points ((x, y) and $(x, p - y)$), hunting-and-pecking requires a deterministic way to choose the output. Here, Dragonfly relies on the Least Significant Bit (LSB) of the seed value (*i.e.,* its parity) that generated the x-coordinate, to determine the compression format of `set_compressed_coord` (Listing 3.1, line 14).

Note that the seed is computed from one secret value and two public ones: the password and MAC addresses of each peer. As a result, any bit of information leaked from the seed can be related to the password, thereby causing a dictionary partitioning attack (cf. chapter 4). Thus, the security of WPA3 also relies on whether the point decompression algorithm is secret-independent regarding the seed parity. Next, we describe the internals of this algorithm as applied for elliptic curves supported by SAE.

Leakage Origin: Compression Format As described in Listing 7.4, the generic decompression algorithm takes three parameters: the x-coordinate of a point, `fmt` as the compression format, and `ec` that stores the curve parameters. The algorithm computes one of the two candidates for y-coordinate using Tonelli-Shanks. Then, it selects the y-coordinate to return based on the value of `fmt` and the parity of y. Because p is an odd prime number, only one y candidate can be even. On a naive implementation, it is easy to notice that the branch line 5 would leak whether the parity of y matches the parity of the compression format. Interestingly, while this line expresses a single condition, it may be broken down into multiple branches. For instance, one can implement it by looking first at the compression format and then processing it differently depending on the parity of y.

Leakage Impact The point decompression method is called with two password-tainted values: the x-coordinate and the compression format. Obviously, we suppose that the attacker does not know either value and cannot make a knowledgeable guess about their parity. However, we suppose that they can guess whether conditional subtraction in line 6 of Listing 7.4 is executed, which implies that they would recover some information about the parity of secret values. The x-coordinate being uniformly distributed on the curve, the parity of y also is. Moreover, since the seed is the output of a cryptographic key-derivation function, it can be seen as the output of a random oracle. Hence, it is plausible to consider that their parity is equal with probability $pr = 1/2$.

This means that an attacker can recover one bit of information if any leakage in line 6 occurs. A second bit of information is recovered if the condition in line 5 is broken into multiple steps: the parity of y or the seed, with the parity equality of both values.

```
1  def bin2bn(buf, n):
2      # Skip leading 0's
3      while (buf[0] == 0)
4          n--
5          buf++
6
7      bn = new_bn()
8      while(n--)
9          add_byte_to_bn(buf, bn)
10
11     return bn
```

Listing 7.5 – Binary to Big Number algorithm. `buf` is a buffer of byte, representing the number, and `n` is the size of the buffer.

Binary to Big Number Conversion

Unlike the vulnerability related to point decompression, the leakage presented below affects hunting-and-pecking and Simplified Shallue-van de Woestijne-Ulas (SSWU)-based hash-to-element.

Leakage Data: Secrets MSB In cryptography, it is common to perform computations on integers not fitting in native types (usually limited to 64 bits on modern architectures). Thus, libraries define a structure called "big numbers", which often boils down to a set of buffers representing the actual value, alongside flags and indexes. However, the manipulated values do not always present as big numbers; they can come in other formats. Thus, they require to be parsed before any computation. Of particular interest, coordinates of the secret point and secret-dependent values shall also be appropriately parsed or converted into the big number structure. Below, we note that the secret values can be leaked if the conversion routine is not secret-independent.

For SAE (*i.e.,* hunting-and-pecking), a coordinate candidate is computed at each iteration as the output of a KDF (Listing 3.1, line 7), that is then converted to a big number. At each iteration, this value is generated like the seed, with additional deterministic processing. Hence, the same consequences follow.

> **Reminder:** In SAE-PT, the password is converted into two integer `u1` and `u2`, using a HKDF with different labels. These integers are then used as input of SSWU to produce two points `P1` and `P2` (section 3.3, Listing 3.2).

For SAE-PT (*i.e.,* SSWU), implementations leak when the coordinates computed by SSWU are set (in the function call line 8 and line 12 of Listing 3.2). The leak may also concern the input to these function calls, denoted `u1` and `u2` in the code sample, as it is the output of the HKDF (line 7 and line 11). All leaked values are computed from the SSID, the password identifier, and the secret password.

Leakage Origin: Optimized Conversion As described in Listing 7.5, we consider the conversion function to take two arguments: the binary buffer and its byte-length. Values in the buffer are considered secret. The routine is straightforward: after some sanity checks (not displayed in the code sample for conciseness), it converts the bytes in `buf` into chunks in `bn` and returns the result.

To avoid processing leading bytes to zero, which does not affect the final value, a common optimization is to skip them before proceeding to conversion, thereby decreasing the buffer size accordingly. An attacker able to guess the number of iterations in the loop line 3 can thus deduce the effective byte-length of the secret.

Leakage Impact The binary conversion function is called several times during the password conversion process. The values we consider are uniformly distributed on the range $[0, 2^{n*8})$ (with negligible bias). Therefore, the k leading bytes are zero with probability $pr = 1/256^k$, leaking $k \times 8$ bits of information.

For legacy SAE (Listing 3.1), three values are underlined: (i) the x-candidates; (ii) the final x-coordinate; and (iii) its corresponding y-coordinate. The MSB of (i) and (ii) may not be not independent, while MSB of (iii) is. Indeed, an attacker can leak information on x-candidates if the observation is different from the one made on the final x-value. This allows the attacker to determine whether the conversion was successful at the first iteration, in addition to the MSB leakage.

In more detail, we distinguish two cases depending on whether the leakage from (i) and (ii) is equal or distinct. As a reminder, each iteration may convert the password successfully with probability $q/2p \approx 0.5$, with p and q being parameters of the curve. In the case of equality, the attacker has to consider these values as equivalent. Both values share a leading byte to zero if they are equal (successful conversion at the first iteration occurs with probability 0.5) or if they are different but share a common leading byte. The probability of such an event is $pr = 0.5 \cdot (1/256 + 1/256^2)$, in which case the attacker would leak 8 bits of information from both observations. Otherwise, if the leaked bits differ, the attacker can infer that both values are different. Thus, they can deduce that the password was not converted during the first iteration. This event occurs with probability 0.5, leaking an additional bit of information. In the end, the attacker learns 9 bits of information with probability $pr = 255/256^2$. Similar reasoning can be applied to subsequent iterations, but the probability of leak quickly decreases and becomes negligible.

As for SAE-PT (Listing 3.2), two points are created from the password. For each point, the two coordinates are converted into big numbers when the resulting point is set. Each coordinate is uniformly distributed on the curve. Hence each call would leak 8 bits with probability 1/256. The same observation applies to the input of SSWU, resulting in three independent leaking values at each call of SSWU. Since it is called twice per run, we expect a $k \times 8$-bit leakage with probability $pr = 6/256^k$.

7.3.2. Vulnerable Implementations

.

> **Takeaway:** Three of the open-source projects we study are vulnerable to at least one of the vulnerabilities, as summarized in Table 7.1.
> Only Apple's CoreCrypto, which re-implements the sensitive functions, does not appear vulnerable.

As stated in subsection 7.3.1, the implementations of vulnerable routines may vary between libraries. Here, we analyze four open-source implementations of SAE and notice that a third-party library usually provides the routines. We evaluated several open-source cryptographic

libraries supported by the studied SAE projects. However, we did not include all the supported libraries. Instead, we only consider those for which elliptic curve operations are implemented since SAE specifically requires their support (with group 19 as a minimal requirement [IEE21]). Moreover, we did not look into any closed source project, such as the one running on Windows.

In this section, we focus on SAE and dicuss SAE-PT later (subsection 7.3.4) for clarity.

Table 7.1. – List of the studied SAE implementations, with leveraged cryptographic libraries. Each • means that the implementation supports the library. Last lines show the average leakage (in bits) from a single session.

		Cryptographic library			
		OpenSSL	WolfSSL	ell	CoreCrypto
Project	hostap	•	•		
	iwd			•	
	FreeRADIUS†	•			
	CoreCrypto				•
Avg. leak `set_compressed_coord`		1	1.5	2	0
Avg. leak `bin2bn`		0.094	0.094	0	0

† FreeRADIUS behaves differently than other Wi-Fi daemon, and leaks more data on `bin2bn` (0.131 bits on average)

The list of evaluated implementations is summarized in Table 7.1. We also note the average number of leaked bits for each handshake execution.

All routines for `set_compressed_coord` and `bin2bn` of OpenSSL[6], WolfSSL[7], and ell[8] are described in Listing 7.6, Listing 7.7 and Listing 7.8 respectively. Below, we go through all these implementations to explore their flaws. We intentionally exclude Apple CoreCrypto[9] routines, although its generic point decompression function also leaks information on the compression format. CoreCrypto, unlike other libraries, provides its own implementation of SAE, where the decompression is re-implemented internally in a secret-independent fashion. In addition, the optimization causing the second vulnerability is not used. This prevents our attacks. We do not claim anything about the SAE running in Apple systems, since this requires reverse engineering efforts to determine whether it leverages CoreCrypto SAE.

For all projects, we compile them using their default settings. Then, we manually confirm the vulnerabilities in the binary and conduct some experiments to assess the Central Processing Unit (CPU) behavior during execution. We extract the relevant routines and measure their execution time while performing a PDA inside the secret-dependent branch. We consistently observe time variation in our measurements. Graphs illustrating these results for `set_compressed_coord` are available in Appendix B, section B.1. Nevertheless, we excluded this leakage vector because we do not consider an attacker capable of timing one single function execution. Moreover, the mild time difference is drowned in the probabilistic execution. We also experimentally confirmed the second vulnerability on OpenSSL (Appendix B, section B.2). Next, we address the actual leakage in each implementation.

[6] https://www.openssl.org
[7] https://www.wolfssl.com/
[8] https://git.kernel.org/cgit/libs/ell/ell.git/
[9] https://developer.apple.com/security/ (OS 2021 release)

```
1  def ec_GFp_simple_set_compressed_coordinates(x, pt, ec):
2      # Compute y candidate
3      y = ...
4      # Comply to input format
5      if ( BN_is_odd(y) != (fmt & 1)):
6          if BN_is_zero(y):
7              # Handle special case, which should never happen with random points
8          BN_usub(y, ec.p, y)
9
10     P = EC_POINT_set_affine_coordinates(x, y)
11     return P
12
13 def BN_bin2bn(buf, n, bn):
14     # Skip leading zero's
15     for ( ; n > 0 && *buf == 0; buf++, n--):
16         continue
17     # Handle special case
18      [...]
19     while (n--):
20         # read the bytes in
21          [...]
22     bn_correct_top(ret)
23     return ret
```

Listing 7.6 – Implementation of the leaking function (`set_compressed_coord` and `bin2bn`) in OpenSSL v1.1.1l. We extracted only the relevant part, simplified the syntax, and renamed variables to save space. Importantly, `fmt` represents the point format, aka the compression type, which can be either 3 or 4 for compressed coordinates.

OpenSSL (Listing 7.6). The point decompression implements the naive approach described in Listing 7.4, which leaks one bit of information for each execution, namely whether the parity of y is equal to the parity of `fmt`. The condition line 6 might also leak if y is zero, which happens with negligible probability for a random point.
The binary conversion skips the leading zero bytes (line 15). In hostap, this function is called on the three different secrets described in subsection 7.3.1, thereby leaking 8 bits of information with probability $p = 1/256 + 0.5 \cdot (1/256 + 1/256^2)$, and 9 bits with probability $p = 255/256^2$ in each session (avg. 0.094 bits per session). In FreeRADIUS, the routine is called on an additional independent secret, leaking more data (avg. 0.131 bits per session).

WolfSSL (Listing 7.7). The point decompression leaks the same information as OpenSSL. However, the `if` statement (line 6-7) is valid if either condition is valid. Hence, upon execution of line 8, an attacker might guess which condition was valid by observing if the function `mp_isodd` was executed once (*i.e.,* y is odd) or twice (*i.e.,* y is even). Here, the attacker would learn the parity of both y and `fmt`, leaking two independent bits of information. The binary conversion leaks the same amount of information as OpenSSL does, because of the optimization in line 15.

ell (Listing 7.8). On one hand, the point decompression leaks the most since it goes through all the point decompression in a switch over `fmt`. The attacker can spy on the executed case to learn `fmt` parity. Then, they can spy on the inner condition to guess the parity of y. Thus, they can learn two independent bits of information for each handshake

```
def wc_ecc_import_point_der_ex(x, ec):
    fmt = x[0]
    # Compute y candidate
    y = ...
    # Comply to input format
    if (mp_isodd(y) and fmt == 3) or
        (not mp_isodd(y) and fmt == 4):
        P.y = mp_mod(y, ec.p)
    else:
        P.y = mp_submod(ec.p, y, ec.p)
    return P

def mp_read_unsigned_bin(bn, buf, n):
    # Skip leading zero's
    while (c > 0 && b[0] == 0):
        c--; b++
    # Handle special case
     [...]
    mp_zero (bn)
    while (n-- > 0):
        # read the bytes in
         [...]
    mp_clamp (bn)
    return MP_OKAY
```

Listing 7.7 – Implementation of the leaking function (`set_compressed_coord` and `bin2bn`) in WolfSSL v5.0.0-stable. We extracted only the relevant part, simplified the syntax, and renamed variables to save space. Importantly, `fmt` represents the point format, aka the compression type, which can be either 3 or 4 for compressed coordinates.

```
def l_ecc_point_from_data(data, fmt, ec):
    memcpy(P.x, data, ec.n)
    if fmt == 4:
        _ecc_compute_y(ec, P.y, P.x)
        if (!(P.y[0] & 1)):
            _vli_mod_sub(P.y, ec.p, P.y, ec.p)
    elif fmt == 3:
        _ecc_compute_y(ec, P.y, P.x)
        if (P.y[0] & 1):
            _vli_mod_sub(P.y, ec.p, P.y, ec.p)
    return P

def _ecc_be2native(bn, buf, n):
    uint64_t tmp[128]
    for (i = 0; i < n; i++)
        tmp[n - 1 - i] = l_get_be64(&buf[i])
    memcpy(dest, tmp, n * 8)
```

Listing 7.8 – Implementation of the leaking function (`set_compressed_coord` and `bin2bn`) in ell v0.47. We extracted only the relevant part, simplified the syntax, and renamed variables to save space. Importantly, `fmt` represents the point format, aka the compression type, which can be either 3 or 4 for compressed coordinates.

execution. The binary conversion, on the other hand, does not leak regarding the MSB of its input.

7.3.3. Evaluation

> **Takeaway:** We designed a Flush+Reload gadget achieving better spatial resolution than the original attack and used it to exploit our vulnerability in a real-world scenario. As a result, we recover reliable execution traces (one bit of information each) in three measurements.

While our identified flaws are exploitable in practice in all studied implementations, we only conduct our experiments in real-world settings against wpa_supplicant (version 2.9) interacting with OpenSSL (version 1.1.1l). This choice was guided to define our experiment settings as close to default users' installations as possible. Indeed, OpenSSL is arguably the most deployed open-source cryptographic library, installed by default in many Linux distributions. Similarly, wpa_supplicant is the default Wi-Fi daemon on Debian-based systems (Ubuntu, Kali, ...), Android, and Red Hat systems for the authentication phase.

Since SAE-PT is not widely deployed yet, we focus on SAE with hunting-and-pecking in this section, and defer our presentation of the impact of our attack on SAE-PT to subsection 7.3.4. As presented in Table 7.1, the vulnerability caused by point decompression is more practical, as it leaks more bits than the one caused by big number conversion. Therefore, we detail our results regarding point decompression, and our test results on the big number conversion vulnerability are summarized in Appendix B, section B.2.

Experimental Setup

All tests were performed on a Dell XPS 13 7390 running on Ubuntu 20.04.2, kernel 5.13.0-39, with an Intel(R) Core(TM) i7-10510U and 16 GB of RAM. All binaries were compiled with gcc version 9.4.0 with all default configurations (optimization included).

We deployed an AP on an Android phone (running Android 11 on a kernel v 4.9.227-perf+) to share network access. In addition, we kept the default configuration on both ends, meaning the key exchange is always performed using group 19, corresponding to P256. Similar results would have been observed with other curves. Our spy process has been implemented using Mastik v0.02 [Yar16] to run the Flush+Reload and PDA threads.

Flush+Reload Gadget

> **Takeaway:** We designed a gadget to overcome the cache line spatial limitation of Flush+Reload and avoid false positives caused by prefetching mechanisms. It creates a race condition between a PDA on the conditional instructions and the prefetching of instructions *after* this line.
> If the conditional instructions are executed, the PDA will slow their execution enough for the attacker to get more cache hits on the following cache lines, prefetched in the execution pipeline.

Failing of the vanilla Flush+Reload. We recall that the attacker needs to successfully determine whether line 8 of Listing 7.6 has been executed. Despite being easily identifiable in

the source code, recovering this leak is challenging since the difference in the control flow only represents a few instructions during the overall execution. Indeed, in the `set_compressed_coord` function, the secret-dependent branch to spy on is quite small, consisting of a single call to an atomic arithmetic operation. Notably, we notice that the probed line is just a few instructions after the branch. This implies that when the execution gets to the branch, the instructions inside the branch might be prefetched to the CPU execution pipeline (thus into the cache) to save some time in case the branch condition is satisfied. This prefetching may also be caused by the spatial locality of the memory lines, since it may be adjacent to the cache line computing the branch condition (more details on CPU optimization are available in section 2.1). Hence, we expect and empirically observe that the probed instructions (line 8) are present in the cache, even without being actually executed by the victim. This results in a high false-positive ratio.

To overcome this issue, we first looked inside the big number subtraction function. Here, the probe is loaded in the cache whenever this function is executed, causing a hit for the attacker for line 8. The problem is that such an atomic mathematical operation is frequently used in cryptographic protocols. Hence, our probe records an overwhelming amount of hits. Moreover, the call to the subtraction function is inconsistent between different executions because of the probabilistic aspect of the hash-to-curve function and the use of random masking.

To summarize, spying on the call of the subtraction function or inside it does not define an accurate distinguisher. Indeed, the former case includes hits even when the spied instructions are not executed, while the latter causes a probabilistic number of hits hindering any finer analysis.

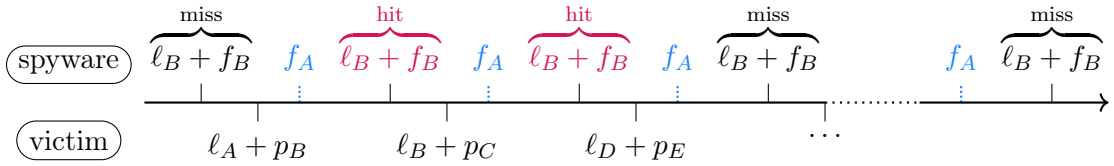
Hacking around the spatial limitations. To overcome this limitation, we designed a gadget probing the next cache line. In the meantime, we perform a PDA to slow down the conditional instructions (usually used as a probe), enough so that the *next* instructions are prefetched into the cache multiple times. Thus, the attacker observes more cache hits if the secret-dependent branch is executed. In our approach, the PDA *creates* the distinguishing behavior that we monitor with Flush+Reload.

With Listing 7.6 as a reference, we performing a PDA on the conditional subtraction (line 8, referred to as cache line A), while probing the instructions in the cache line *after* the branch (line 10, referred to as cache line B). Interestingly, this allows us to obtain a practical distinguisher characterizing the leakage. Below, we unfold the rationality behind this successful attempt.

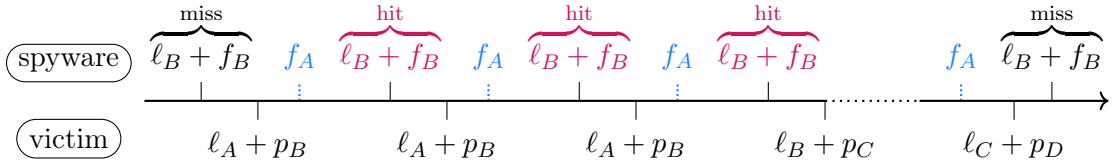
During execution, we observe that the prefetcher loads both the branch instructions and the probe in the execution pipeline (cache lines A and B). At this point, the probed instructions are detected for the first time by the attacker. We stress that after this hit, the probe is flushed by the spyware before its next observation. Here, we expect to observe two behaviors with regard to the PDA on the conditional subtraction. These behaviors are depicted in Figure 7.2.

First, if the condition in line 5 is invalid (Figure 7.2a), the PDA does not have any impact, as the execution will continue normally. We may get a second hit on the probe because it is reloaded when the function call returns in line 10.

Second, if the condition verifies (Figure 7.2b), the increased cache-misses caused by the PDA in line 8 slows down the execution. In fact, cache line A contains multiple assembly instructions to be executed by the CPU, but the PDA repeatedly evicts them before they can be executed. This makes the CPU fetch them back, thereby prefetching the next instructions



(a) Cache-related operations when the condition is false.



(b) Cache-related operations when the condition is true.

Figure 7.2. – Memory fetching and flushing by the spyware (on top annotations) and the victim (bottom annotations). ℓ represents a loading operation, f represents a flush and p denotes a prefetching. Indexes represent the target cache-line, cache-line A contains the conditional instructions, cache-line B contains the probe and C, D and E represent following instructions. Blue annotations are from the PDA, cache hit on the probe are displayed in red.

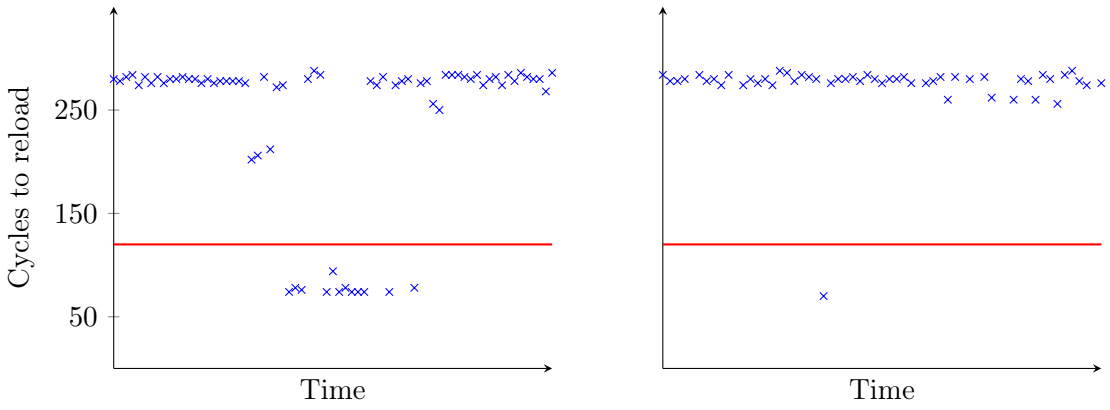
simultaneously. This additional processing time is enough to reach the temporal resolution of Flush+Reload, so that new measurements can be achieved within this interval. For each measurement, the probe is first flushed and then reloaded after the predetermined idle. In the meantime, whenever the prefetcher reloads the probe, the attacker gets new hits for each measurement cycle.

In a nutshell, our distinguisher is defined as follows: while probing line 8, we consistently obtain more hits when line 6 is executed. The reasons are twofold. First, the PDA slows down the execution of the conditional branch by evicting its instructions out of the cache. Second, some CPU optimization gets the probe back in the cache due to the spatial proximity with the conditional branch. As a side note, the number of cache hits may be used to define a confidence coefficient to prune false positives and avoid poisoning our dataset if we can only gather a limited number of traces.

We assume this behavior to be caused by prefetching mechanisms but did not identify the exact cause. Interestingly, disabling the four CPU prefetchers documented in [Int21] (streamer, Spatial, Data Cache Unit, and Instruction Pointer-based) using Model-Specific Register (MSR) 0x14a did not change the outcome. This suggests either a different prefetcher affecting the L1i or another mechanism involved. Isolating the exact component responsible for this characteristic behavior would need more experimentations and is left as future work (see Part III).

Practical Attack Against wpa_supplicant

We target the OpenSSL function `ec_GFp_simple_set_compressed_coordinates`, represented in Listing 7.6. We recall that, in SAE, this function is executed at the end of the password conversion (Listing 3.1, line 14), after going through the 40 iterations of the hunting-and-pecking.



(a) `y` $\not\equiv$ `fmt` mod 2 **(b)** `y` $\equiv$ `fmt` mod 2

Figure 7.3. – Representation of a measurement when the branch is executed (left) and not executed (right). The red line represents the threshold: each blue cross under the line means that the victim has loaded the probe in the cache.

Applying our gadget, we probed the call to the function `EC_POINT_set_affine_coordinates` with Flush+Reload while performing a PDA on both the call to `BN_usub` and its internal instructions. The PDA aims to increase the latency of `BN_usub` by forcing the execution pipeline to fetch our probe multiple times. On the resulted binary, the monitored instructions are located 101 bytes from the call to `BN_usub`. This means that the probe, and the conditional instructions, are at most one cache line apart and possibly on adjacent cache lines (considering 64-bytes lines). We suspect that such spatial proximity triggers some CPU optimization to fetch in the next instructions. Indeed, we notice that an execution where the parity of `fmt` and `y` is different results in more hits.

In practice, we observed an increase of cache hits by a factor of five to ten (going from a couple of hits to about ten or more). This difference is easily recognizable, as shown in Figure 7.3. Here, the threshold (horizontal red line) between a cache-hit and a cache-miss is empirically defined by measuring the average loading latency of each action. We can observe significantly more hits (blue cross under the red threshold) when the call to `EC_POINT_set_affine_coordinates` is executed (*i.e.,* leaking that the parity is not equal).

Performance and Accuracy

As with any cache-based attacks, our measurements are susceptible to system noise and CPU optimizations. Therefore, multiple spied observations are required for the same password and MAC addresses to get dependable results. Below, we describe our settings to assess the accuracy of our attack. We conduct our evaluation with 20 different passwords. For each run, we collect 20 traces while varying one MAC address. This gives a total of 400 samples.

Our findings show that our technique to overcome the spatial limitations of Flush+Reload is rewarding. Indeed, by discarding low confidence traces, we could exploit 365/400 samples, with only ten miss-predictions, from only two measurements per sample. Three measurements with the same password and MAC addresses are enough to achieve 100% of usability, and 100% of accuracy. Our dataset is available in the PoC repository of our attack[10].

[10] https://anonymous.4open.science/r/artifact_dragondoom-692C/

Then, we continue our study by computing the average number of hits for three measurements. Henceforth, we refer to this value as the *execution trace*, or simply as a *trace*. As explained previously, in the case of OpenSSL, each trace reveals one bit of information about the password.

7.3.4. Impact on SAE-PT

Since SAE-PT does not involve a call to the point decompression method, the first vulnerability does not apply. ell is not affected since it does not provide conversion optimization. FreeRADIUS does not implement SSWU. However, hostap with both OpenSSL and WolfSSL suffers from the same leakage with the second vulnerability.

Indeed, hostap calls the `bin2bn` routine on six independent secrets along the hash-to-element method: the input of SSWU and both point coordinates output (SSWU is repeated twice, hence the six occurrences). As described in subsection 7.3.1, this would allow the attacker to get $k \times 8$ bits of information with probability $6/256^k$.

Interestingly, the identifier is not attacker-controlled since it is set in a configuration file and not shared on the network, and the SSID cannot be changed without the client noticing. Hence, the attacker cannot arbitrarily modify public values involved in the computation. This significantly limits their ability to aggregate information by repeating the measurements. The only element they may vary is the curve used for the derivation.

The standard mentions up to three different curves, and both WolfSSL and OpenSSL support them. Hence, the attacker may repeat this measurement up to three times, meaning they could leak 8 bits of information with a probability $p = 18/256$, or in 7% of the cases. This leakage expands to 16 bits in approximately 0.03% of the cases, which is still relevant considering the broad deployment of WPA3.

On the other hand, ell is not affected since it does not provide conversion optimization. FreeRADIUS does not implement SSWU.

7.3.5. Mitigations

The most common way to mitigate this type of vulnerability is to address them at the application level with a secret-independent implementation. This approach benefits from two major upsides: (i) the patch is often simple; and (ii) the overhead is minimal and limited to the particular application. It is important to highlight that a particular aspect of our vulnerabilities lies in an uncommon interaction between SAE and its cryptographic provider. Indeed, in all the studied libraries, the function setting a point coordinate never handles the compression format as a secret.

Regarding point decompression, the mitigation is similar in all the affected projects. Both potential point coordinates must be computed to ensure a secret-independent control flow, followed by a constant-time selection. After our disclosure, this solution has been implemented by hostap, FreeRADIUS, and iwd. It is worth noting that the execution of the original routines `set_compressed_point_coordinate` in OpenSSL, WolfSSL, and ell are still dependent on the compression format.

Unlike point decompression, we included the cryptographic libraries in our disclosure for the big number conversion vulnerability, namely OpenSSL and WolfSSL. This is because such libraries encapsulate the definition of the big number structure. Thus, any related code is reasonably expected to be patched by the concerned libraries. Both OpenSSL and WolfSSL

acknowledged that their big number operations are leaky but refused to patch. Indeed, they argued that it is upon developers' responsibility to avoid calling these functions with secret-dependent values.

7.4. Comparative Analysis

7.4.1. Concept of the Attacks

The original Dragonblood attacks [VR20] presented a side-channel on the password conversion. They recover information on the password by observing the duration of the first iteration of the password conversion. Since the implementation was not secret-independent, a successful conversion would need more time to process the additional instructions. A successful conversion occurs with a probability of 0.5. Hence, this attack provides a fixed amount of leakage on the password.

In section 7.2, we present a similar attack on implementations that did not properly mitigate Dragonblood. We improved the initial attack by targeting more specific instructions to extract more information from each observation. Namely, instead of only learning whether the successful conversion occurs during the first iteration, we get the exact corresponding iteration. Considering the conversion probability at each iteration, this doubles the amount of obtained leakage.

In section 7.3, we present attacks that differ from previous works in multiple aspects. First, we identify a *new leakage source*. Previous attacks leaked information about the conversion iteration, and they were patched by making the workflow of each iteration of the loop secret-independent. Our attack instead targets generic functions from the called cryptographic library. Second, our vulnerabilities provide a wider *scope* for an attacker, impacting not only all previously patched implementations but also the recently supported SAE-PT.

7.4.2. Performance and Efficiency

> **Reminder:** We use the signal-to-noise metric to compare the side-channel efficiency (chapter 4). That is the ratio between the leakage for a fixed set of parameters and the measurements necessary for this leakage to be reliable.

Table 7.2 sums up the average number of measurements (*i.e.,* SAE handshakes spied on) needed to prune all invalid passwords of various dictionaries with high probability ($p > 0.95$), as described in section 4.2. We stress that we consider not only the number of bits leaked by a handshake, but also the number of measurements required to exploit it reliably. This is represented by the signal-to-noise ratio in the last line of Table 7.2.

For [VR20], presenting a collection of attacks, we only consider the cache-attack leaking whether the first iteration of the conversion loop successfully finds an appropriate x-coordinate. Authors needed to repeat their measurements up to 20 times to get an exploitable leak. The theoretical vulnerability leaks 2 bits on average, but the provided PoC only leaks one bit. Table 7.2 only takes into account their implemented attack, since implementing their theoretical attack may need more measurements. Our results reveal that our attacks are 4 and 6.667 times more efficient.

In section 7.2, we expanded on [VR20] to refine the attack and leak more accurate information, with fewer measurements. We were able to leak the exact iteration corresponding to the

successful password conversion. This additional leak allowed the attacker to obtain about 2 bits of information on average. However, we still need ten microarchitectural measurements for a reliable execution trace.

The attack introduced in section 7.3, in addition to exploiting an unstudied leakage vector in WPA3, achieves better efficiency: it leaks one bit of information per trace, with only three measurements needed for each trace. It is worth noting that this is only about exploiting hunting-and-pecking in OpenSSL. Better results are expected for WolfSSL and ell since they may leak more bits. Moreover, the case of SSWU is quite different for two reasons. First, the attacker can only obtain one execution trace while spying on SAE-PT. Second, a trace may leak more bits with lower probability (refer to subsection 7.3.1). Consequently, this vulnerability is considered less practical, but it still provides an important insight into the difficulty of providing secret-independent implementations even for constant-time algorithms by design.

Table 7.2. – Comparison of the number of the measurements needed to prune all wrong passwords of various dictionaries for our attacks and previous work (with probability $p > 0.95$). The last line shows the signal-to-noise ratio of the attacks (the higher the better).

	[VR20]	Section 7.2	Section 7.3
rockyou ($1.4 \cdot 10^7$)	580	**150**	**87**
CrackStation ($3.5 \cdot 10^7$)	600	**150**	**90**
HaveIBeenPwned ($5.5 \cdot 10^8$)	680	**170**	**102**
8 characters ($4.6 \cdot 10^{14}$)	1060	**270**	**159**
Overall efficiency ($\mathcal{D}$)	0.05	**0.2**	**0.33**

7.5. Discussion

Lingering Vulnerabilities. After the original Dragonblood publication, implementations received various patches and dropped the support of Brainpool curves. However, the primary source of vulnerabilities, the probabilistic hash-to-group function, was still unchanged despite the standards update. Even more worrying, despite proper branch-free implementations being publicly available with negligible overhead, most implementations did not patch the secret-dependent control flow of the password derivation. We believe the lack of patch is strongly related to the lack of PoC dedicated to specific implementations. Dragonblood only describes the attack for hostapd, which has been fixed.

The attack we describe in section 7.2 illustrates the risk to users when cryptographic software developers dismiss a widely potential attack. Unfortunately, this is the prevailing approach for security vulnerabilities, but we show that this approach is fraught with danger for standards like WPA3. Hopefully, the Wi-Fi Alliance dropped their ad-hoc mitigations for a constant-time algorithm by design that does not rely on savvy developers to provide secure implementations.

Does SSWU Worth it? SSWU is proposed as a superior alternative in WPA3 both in terms of efficiency and security. Indeed, SSWU is a deterministic mapping, which means that

it does not suffer from the inherent secret-dependence issue of a probabilistic approach such as hunting-and-pecking. Moreover, the ongoing standardization process of hash-to-curve functions provides a straightforward and secret-independent implementation of SSWU [FSS+22].

Nevertheless, in section 7.3, we show that vulnerabilities still sneak into deployed implementations of SSWU. The identified vulnerability is less exploitable in practice, but we still provide important insight regarding the use of third-party libraries. Indeed, although capital, secret-independent design is not enough if the low-level operations (*e.g.,* arithmetic ones) are leaky. Despite our vulnerability, we still believe that SSWU offers better security to settle the cat-and-mouse Side-Channel Attack (SCA) concerns about Dragonfly.

Limitations of the hack-and-patch approach. As convenient as it might seem, providing quick and simple patches is prone to error and only provides relative security (as demonstrated by these vulnerabilities, despite previous mitigations). It is time to bring a sustainable, long-term solution to the large class of vulnerabilities that are side-channel attacks. Fortunately, researchers have made consistent progress computer-aided cryptography within the last decade [BBB+21]. To eliminate implementation-related side-channels, two approaches stand out from the current solutions.

First, one might keep the benefit of existing implementations and perform an extensive analysis of the code base. Then, to provide decent guarantees, the minimal criterion would be to provide *sound* results, meaning only secure programs are deemed secure. As it turns out, formally verifying large implementations is daunting if we must dig deeper each time an abstraction layer is defined. Namely, a tool providing complete coverage, formal guarantees, and scalability is yet to be seen. However, even sacrificing the formal aspect for gaining scalability, computer-aided verification would help to catch most side channels early. The academic community is prolific in this field, and numerous solutions are provided, some specifically designed to be straightforward to use [Lan10; Nei; HEC20]. This raises the question of why is today's cryptographic software not free of timing-attack vulnerabilities. We investigate and try to answer this question in chapter 8.

Second, instead of verifying existing implementations, we can tackle the issue from the other end and *create* formally verified implementations. We describe this solution in chapter 9 and provide a verified implementation of the cryptographic operations used in SAE(-PT).

Usage and Usability of Constant-Time Verification Tools

8

In this chapter, we set out to understand why is today's cryptographic software not free of timing-attack vulnerabilities, despite numerous academic contributions in that direction. To this end, we conduct a mixed-methods online survey with 44 developers of 27 popular cryptographic libraries. Through this survey, we track down the origin of the persistence of such side-channel attacks.

In section 8.1, we motivate the need for constant-time implementations, set the current state of verification tools, and properly define the questions we aim to answer. In section 8.2, we give an overview of timing attacks history, define the scope of our survey (the tools and projects we included), and give some insight on related works. In section 8.3, we describe our methodology to conduct this study, including setting the survey structure and analyzing the results. Then, in section 8.4, we present a synthesis of our results, that we discuss in section 8.5.

Takeaway:

This contribution supports the following conclusions:

- ➤ Developers are aware of the danger of side channels, but not all of them know about existing tools to help them prevent these vulnerabilities.
- ➤ There is a leaky pipeline in the tool adoption: despite a prolific research field, few tools are used by developers.
- ➤ The availability and usability of supportive tools need to be improved in order to lower the burden on cryptographic library developers.

Contrary to previous chapters, this chapter is entirely self-contained and does not require reading Part I. Instead, a specific background dedicated to this chapter is available in section 8.2. However, skimming through previous chapters (chapter 6 and chapter 7) may help to grasp the danger of timing attacks, and the need for sustainable solutions.

Contents

8.1. Introduction **109**
8.2. Background & Related Work **110**
8.2.1. Attacks 110
8.2.2. Tools included in the survey 112
8.2.3. Libraries included in the survey 114
8.2.4. Additional Related Work 114
8.3. Methodology **116**
8.3.1. Study Procedure 117
8.3.2. Survey Structure 117
8.3.3. Coding and Analysis 119
8.3.4. Data Collection and Ethics 120
8.3.5. Limitations 120
8.3.6. Data cleaning & Presentation 120
8.4. Results **121**
8.4.1. Survey Participants 121
8.4.2. Answering Research Questions 122
8.5. Discussion **133**
8.5.1. Tool developers 133
8.5.2. Compiler writers 134
8.5.3. Cryptographic library developers 134
8.5.4. Standardization bodies 135

8.1. Introduction

> **Reminder:** *Timing-attacks* are a class of side-channel attacks exploiting a timing difference in the processing, whether it is on the overall operation or on internal manipulations (chapter 1). This can be caused by a secret-dependent control flow (*e.g.,* when an implementation branches on a value derived from a secret), or a secret-dependent memory access (*e.g.,* when an array is indexed with a secret value).

Although Kocher first described timing attacks in 1996 [Koc96], they continue to plague implementations of cryptographic libraries. At the same time, and most importantly for this chapter, we know how to systematically protect against timing attacks. Kocher already described the basic idea of such systematic countermeasures in 1996 [Koc96]: we need to ensure that all code takes time independent of secret data. It is important not just to consider the total time taken by some cryptographic computation, but to ensure that this property holds for each instruction. This paradigm is known as *constant-time*[1], or secret independence, and is usually achieved by ensuring that

- there is no data flow from secrets into branch conditions;
- addresses used for memory access do not depend on secret data; and
- no secret-dependent data is used as input to variable-time arithmetic instructions (such as, *e.g.,* `DIV` on most Intel processors or `SMULL`/`UMULL` on ARM Cortex-M3).

Constant-timeness is no panacea, and the above rules may not be sufficient on some microarchitectures or in the presence of speculative execution. However, essentially all timing-attack vulnerabilities found so far in cryptographic libraries (including the one we described in chapter 6 and chapter 7) could have been avoided by following these rules. For this reason, the notion of constant time has grown in importance in standardization processes and recent cryptographic competitions. For instance, in the context of the ongoing Post-Quantum Cryptography Standardization project, the National Institute of Standards and Technology has stated in their Call for Papers [NIS16]:

> *"Schemes that can be made resistant to side-channel attack at minimal cost are more desirable than those whose performance is severely hampered by any attempt to resist side-channel attacks. We further note that optimized implementations that address side-channel attacks (*e.g., *constant-time implementations) are more meaningful than those which do not."*

Protection against side-channel attacks, including timing attacks, is also routinely included as a requirement for Common Criteria certification and a part of the newly approved FIPS 140-3 certification scheme [ABB+20b].

Programming highly optimized code that is also constant-time can be very challenging. However, we know how to verify that programs are constant-time. This was first demonstrated by Adam Langley's ctgrind [Lan10], developed in 2010, the first tool to support the analysis of constant-timeness. A decade later, there are now more than 30 tools for checking that code satisfies constant-timeness or is resistant against side-channels [Jan21; BBB+21]. These

[1] The term constant-time, often referred to as CT, is a bit of a misnomer, as it does not refer to Central Processing Unit (CPU) execution time but rather to a structural property of programs. However, it is well-established in the cryptography community.

tools differ in their goals, achievements, and status. Yet, they collectively demonstrate that automated analysis of constant-time programs is feasible; for instance, a 2019 review [BBB+21] lists automatic verification of constant-time real-world code as one achievement of computer-aided cryptography, an emerging field that develops and applies formal, machine-checkable approaches to the design, analysis, and implementation of cryptography.

Based on this state of affairs, one would expect that timing leaks in cryptographic software have been systematically eliminated, and timing attacks are a thing of the past. Unfortunately, this is far from true, so in this chapter, we set out to answer the question:

Why is today's cryptographic software not free of timing-attack vulnerabilities?

More specifically, to understand how real-world cryptographic library developers think about timing attacks and the constant-time property, as well as constant-time verification tools, we conducted a mixed-methods online survey with 44 developers of 27 popular cryptographic libraries/primitives[2]. Through this survey, we track down the origin of the persistence of timing attacks by addressing multiple sub-questions:

RQ1: **(a)** Are timing attacks part of threat models of libraries/primitives? **(b)** Do libraries and primitives claim resistance against timing attacks?

RQ2: **(a)** How do libraries/primitives protect against timing attacks? **(b)** Are libraries and primitives being verified/tested for constant-timeness? **(c)** How often/when is this done?

RQ3: **(a)** What is the state of awareness of tools that can verify constant-timeness? **(b)** What are the experiences with the tools?

RQ4: Are participants inclined to hypothetically use formal-analysis-based, dynamic instrumentation, or runtime statistical test tools, based on tool use requirements and guarantees?

RQ5: What would developers want from constant-time verification tools?

We find that, while all 44 participants are aware of timing attacks, not all cryptographic libraries have verified/tested resistance against timing attacks. Reasons for this include varying threat models, a lack of awareness of tooling that supports testing/verification, lack of availability, as well as a perceived significant effort in using those tools (see Figure 8.1). We expose these reasons and provide recommendations to tool developers, cryptographic libraries developers, compiler writers, and standardization bodies to overcome the main obstacles toward a more systematic protection against timing attacks. We also briefly discuss how these recommendations extend to closely related lines of research, including tools for protecting against Spectre-style attacks [KHF+19].

8.2. Background & Related Work

8.2.1. Attacks

In 1996, Kocher introduced the concept of *timing attacks* as a means to attack cryptographic implementations *"by carefully measuring the amount of time required to perform private key*

[2]We refer to both as "libraries" for readability.

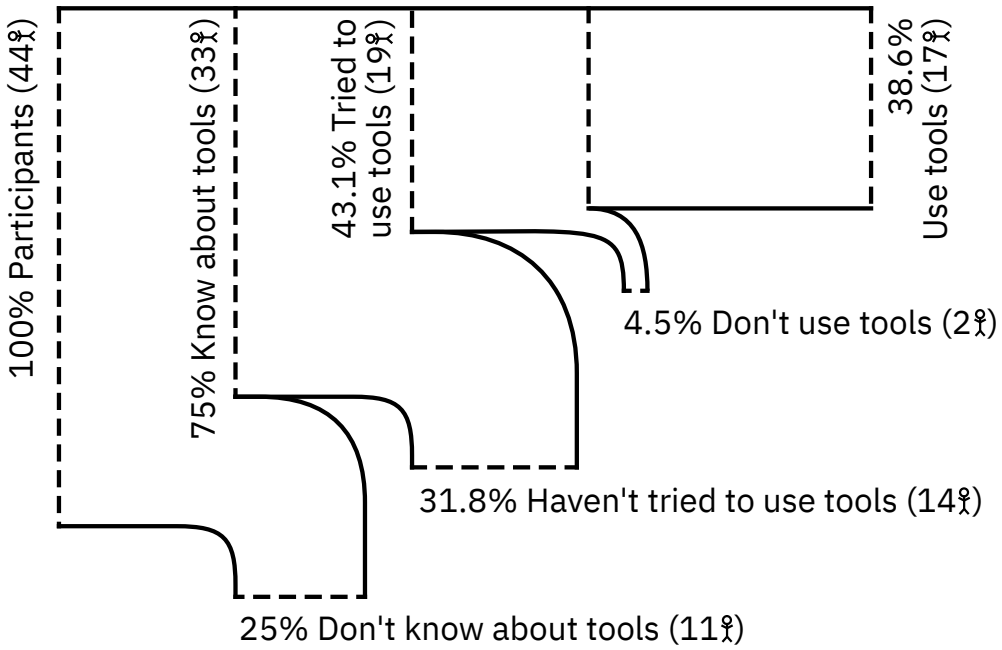


Figure 8.1. – Leaky pipeline of developers' knowledge and use of tools for testing or verifying constant-timeness.

operations" [Koc96]. He described successful timing attacks against implementations of various building blocks commonly used in asymmetric cryptography like modular exponentiation and Montgomery reduction against RSA and DSS. Since this seminal paper, timing attacks have been further refined and continued to plague implementations of both asymmetric and symmetric cryptography. Successful timing attacks are way too numerous to list, so we focus on a few relevant examples.

In 2002, Tsunoo *et al.* [TTMM02; TSS+03] were the first to present attacks exploiting cache timing to break symmetric cryptography (MISTY1 and DES); they also mentioned a cache-timing attack against AES. Details of cache-timing attacks against AES were first presented in independent concurrent work by Bernstein [Ber05] and by Osvik, Shamir, and Tromer [OST06]. In 2003, D. Brumley and Boneh showed that timing attacks can be mounted remotely by measuring timing variations in response times of SSL servers over the network [BB03]. Canvel *et al.* showed in 2003 how to recover passwords in SSL/TLS channels using padding oracle attacks [CHVV03]. In 2011, B. Brumley and Tuveri showed that such remote attacks are still possible [BT11]; *i.e.,* that the underlying weaknesses in the OpenSSL library had not been suitably fixed. SSL libraries continued to be the target of timing attacks; examples include the "Lucky 13" attack by AlFardan and Paterson, which exploits timing variation in the processing of padding in the CBC mode of operation in multiple common SSL/TLS libraries [FP13] similar in principle to the paper by Canvel *et al.* [CHVV03]. In 2015, Albrecht and Paterson presented a variant of the attack targeting Amazon's s2n implementation of TLS [AP16]. In 2016, Yarom, Genkin, and Heninger presented the "CacheBleed" attack, which showed that the "scatter-gather" implementation technique recommended by Intel [Bri11] and implemented in OpenSSL as cache-timing attack countermeasure, is insufficient to thwart attacks [YGH16]. In the same year, Kaufmann *et al.* showed that even carefully implemented C code may be translated to binaries that are vulnerable to timing attacks [KPVV16]. In

chapter 6, we presented a timing attack affecting OpenSSL's modular exponentiation, making numerous Secure Remote Password protocol (SRP) implementations vulnerable. Even more recently, in chapter 7, we demonstrated that even after multiple assessement, side-channel still find their way in Wi-Fi Protected Access (WPA)3 implementations.

We conclude this paragraph with a few attacks related to certification and standardization. Certification schemes such as Common Criteria often require certified products to have countermeasures to a range of side-channel attacks, including timing attacks. However, certified hardware did not avoid being a target of timing attacks, as shown by the recent Minerva group of vulnerabilities in Elliptic Curve Digital Signature Algorithm (ECDSA) implementations, including a Common Criteria certified smartcard [JSSS20]. In recent years, various timing attacks have been proposed against implementations of post-quantum cryptography (PQC), including attacks against the BLISS signature scheme used in the strongSwan IPsec implementation [BHLY16; PBY17; BBE+19; TW21] and attacks against candidates in NIST's PQC standardization effort [PT19; WBB+20; GJN20; MUTS22].

Despite all these academic timing attacks, practitioners often question their practical exploitability. Security Audit companies try to catch timing vulnerabilities in software [MS14]. However, they make the following statements: *"Even though there is basic awareness of timing side-channel attacks in the community, they often go unnoticed or are flagged during code audits without a true understanding of their exploitability in practice."*

8.2.2. Tools included in the survey

We provide a brief overview of the tools considered in our survey. We classify tools according to their broad approach: runtime statistical tests, dynamic-instrumentation based, or formal-analysis based. Our approach, as well as our choice of included tools, is based on an earlier paper [BBB+21], but amended with tools some authors know to be in current use.

Broadly speaking, statistical test tools [RBV17] compute the execution time of a large number of runs of the target program and verify whether secret data influences the execution time. These tools are generally easy to install and run, even at scale, and operate on executable code, ruling out the possibility of compiler-induced violations of the constant-time policy. However, they only provide weak, informal guarantees.

In contrast, dynamic instrumentation based tools [DBR20; BPT17; WWL+17; BSBP18; Lan10; HEC20; WSBS20; WZS+18; UCSa; Tea20; WMES18b; Nei; Cuo] instrument programs to track how information flows during (concrete or symbolic) execution of programs. They are generally reasonably easy to install and to use, even at scale, and can be implemented at source, intermediate, or assembly levels, and provide formal guarantees. However, as with all tools based on dynamic techniques, these guarantees are generally limited; for instance, dynamic analysis of loops may be unsound (*i.e.,* miss constant-time violations).

Finally, formal-analysis-based tools [ABPV13; AGH+17; DFK+13; BGA+16; WRP+19; CSJ+19a; RPA16; KMO12; ACE+18; CFD17; BBC+14] provide strong guarantees that programs do not violate constant-timeness; in addition, some tools are precise, in that they only reject programs that violate constant-timeness. Their other criterion is soundness, which ensures the absence of constant-time violations. However, these tools are often implemented at source or intermediate levels, frequently require user interaction, and are sometimes hard to install or use at scale.

Table 8.1 presents some key tools and summarizes their main characteristics. Since our focus is not an in-depth technical comparison of the features of the tools, we deliberately

keep descriptions simple and only consider their target and whether they provide some formal guarantees (No, Partial, Yes, Other). For the cognizant, "Partial guarantees" cover tools that perform dynamic analysis, whereas "Guarantees" cover tools that are sound and detect all constant-time violations; in particular, our classification does not reflect if tools are precise. Even for such coarse criteria, classification is sometimes challenging, so we err on the generous side. Finally, we tag tools as "Other" if they establish another property than constant-time; comparing these properties with constant-time is often tricky, so we choose not to qualify the difference.

Table 8.1. – Classification of tools included in the survey.

Tool	Target	Techn.	Guarantees
ABPV13 [ABPV13]	C	Formal	●
Binsec/Rel [DBR20]	Binary	Symbolic	◐
Blazer [AGH+17]	Java	Formal	●
BPT17 [BPT17]	C	Symbolic	◐
CacheAudit [DFK+13]	Binary	Formal	■
CacheD [WWL+17]	Trace	Symbolic	○
COCO-CHANNEL [BSBP18]	Java	Symbolic	●
ctgrind [Lan10]	Binary	Dynamic	◐
ct-fuzz [HEC20]	LLVM	Dynamic	○
ct-verif [BGA+16]	LLVM	Formal	●
CT-WASM [WRP+19]	WASM	Formal†	●
DATA [WZS+18; WSBS20]	Binary	Dynamic	◐
dudect [RBV17]	Binary	Statistics	○
FaCT [CSJ+19a]	DSL	Formal†	●
FlowTracker [RPA16]	LLVM	Formal	●
haybale-pitchfork [UCSa]	LLVM	Symbolic	◐
KMO12 [KMO12]	Binary	Formal	■
MemSan [Tea20]	LLVM	Dynamic	◐
MicroWalk [WMES18b]	Binary	Dynamic	◐
SC-Eliminator [WGSW18]	LLVM	Formal†	●
SideTrail [ACE+18]	LLVM	Formal	■
Themis [CFD17]	Java	Formal	●
timecop [Nei]	Binary	Dynamic	◐
tis-ct [Cuo]	C	Symbolic	◐
VirtualCert [BBC+14]	x86	Formal	●

Targets: LLVM - intermediate representation, DSL - domain-specific language, WASM - Web Assembly
Technique: † - also performs code transformation/synthesis
Guarantees: ● - sound, ◐ - sound with restrictions, ○ - no guarantee, ■ - other property

While the CoCo-Channel authors wrote [BSBP18]: *"We also evaluate CoCo-Channel against two recent tools for detecting side-channel vulnerabilities in Java applications, Blazer and Themis. Neither are publicly available[...]"*, their tool was not found by us either.

We do not claim our list to be comprehensive, especially in this currently active field of

research. In particular, we did not ask about Constantine [BDQG21], Pitchfork-angr [UCSb], Cachefix [CR18], and ENCoVer[BDG12], just to name a few.

8.2.3. Libraries included in the survey

Cryptographic libraries have diverse threat models, but with their usual use in protocols like TLS and connected applications often running on shared hardware, resistance against timing attacks is an important property. In our survey, we invited developers of all widely used TLS libraries and other smaller but popular libraries and relevant primitives. We focused on libraries implemented in C/C++ as it is the target language of most tools and the most used language for cryptographic libraries. However, we included libraries implemented in Java, Rust, and Python if some tools can analyze them or they contain parts implemented in C.

Our choice of libraries is underpinned by our knowledge of them and by quantitative data of user and developer numbers. We included some newer primitives not (yet) fulfilling this criterion to complement the answers given by the first group. Nemec *et al.* [NKS+17b] gave numbers for OpenSSL: *"The prevalence of OpenSSL reaches almost 85% within the current Alexa top 1M domains and more than 96% for client-side SSH keys as used by GitHub users."* We only included libraries with an open development model to allow us to get data for our recruiting choice.

Table 8.2 contains a list of libraries included in the survey and whether at least one of their developers participated in our survey. The table also lists the actions that the libraries perform in their Continuous Integration (CI) pipelines. We draw this information from documentation and the public CI pipelines of the libraries. One author extracted this, and a second author double-checked, with disagreements discussed and resolved.

8.2.4. Additional Related Work

Having already discussed timing attacks and tools for constant-time analysis, we briefly cover other related work.

Foundations of constant-time programming. Constant-time programming is supported by rigorous foundations. These foundations establish that programs are protected against passive adversaries observing program execution. However, Barthe *et al.* [BBC+14] show that constant-time programs are protected against system-level adversaries that control the cache (in prescribed ways) and the scheduler. Recently, these foundations have been extended to reflect microarchitectural attacks [CBS+19; CDM+22; GKM+20; CDG+20]. In parallel, many tools are being developed to prove that programs are speculative-constant-time, a strengthening of the constant-time property which offers protection against Spectre [KHF+19] attacks. We expect that many of our work's takeaways apply to this novel direction of work.

High-assurance cryptography. High-assurance cryptography is an emerging area that aims to build efficient implementations that achieve functional correctness, constant-timeness, and security. High-assurance cryptography has already achieved notable successes [BBB+21]. The most relevant success in the context of this work is the EverCrypt library including HACL* and ValeCrypt [ZBPB17; PPF+20; PBP+20], which has been deployed in multiple real-world systems, notably Mozilla Firefox and Wireguard VPN. The EverCrypt library is formally verified for constant-timeness (and functional correctness). However, the library is conceived

Table 8.2. – Libraries and primitives included and the actions they perform in their public continuous integration pipelines.

Library	**Particip.**	**Continuous integration**			
		Build	Test	Fuzz‡	CT test
OpenSSL	✔	✔	✔	✔	✘
LibreSSL	✔	✔	✔	✔	✘
BoringSSL	✔	✔	✔	✔	✔
BearSSL	✔	✔	✔	✘	✘
Botan	✔	✔	✔	✔	✔
Crypto++		✔	✔	✘	✘
wolfSSL	✔	✔	✔	✔	✘
mbedTLS	✔	✔	✔	✔	✔
Amazon s2n	✔	✔	✔	✔	✔
MatrixSSL			No public CI		
GnuTLS	✔	✔	✔	✔	✘
NSS	✔	✔	✔	✔	✘
libtomcrypt	✔	✔	✔	✘	✘
libgcrypt	✔		No public CI		
Nettle	✔	✔	✔	✔	✘
Microsoft SymCrypt	✔	✔	✔	✔	✘
Intel IPP crypto			No public CI		
cryptlib	✔		No public CI		
libsecp256k1	✔	✔	✔	✔	✔
NaCl	✔		No public CI		
libsodium	✔	✔	✔	✔	✘
monocypher	✔	✔	✔	✔	✘
BouncyCastle*	✔	✔	✔	✘	✘
OpenJDK		✔	✔	✘	✘
dalek-cryptography†		✔	✔	✘	✘
ring†		✔	✔	✔	✘
RustCrypto†	✔	✔	✔	✘	✘
rustls†	✔	✔	✔	✔	✘
python-ecdsa	✔	✔	✔	✘	✘
micro-ecc			No public CI		
tiny-AES-c	✔	✔	✔	✘	✘
PQCrypto-SIDH	✔	✔	✔	✘	✔
csidh	✔		No public CI		
constant-csidh-c-implementation	✔		No public CI		
ARMv8-CSIDH			No public CI		
SPHINCS+		✔	✔	✘	✘
Total = 36	27 (75%)	27 (75%)	27 (75%)	16 (44%)	6 (17%)

* Java; † Rust; ‡ Includes being fuzzed by OSS-Fuzz or cryptofuzz.

as a drop-in replacement for existing implementations. Despite relying on an advanced infrastructure built around the F* programming language, this work does not explicitly target open-source cryptographic library developers as potential users of their infrastructure. We cover this aspect later, in chapter 9. Other projects that enforce constant-time by default, such as Jasmin [ABB+20a; ABB+17] or FaCT [CSJ+19a], target open source cryptographic library developers more explicitly, but rely on domain-specific languages, which may hinder their broad adoption. In contrast, we focus on tools that do not impose a specific programming framework for developers.

Human factor research. Researchers have tried to answer the question of why cryptographic advances do not necessarily reach users. In a 2017 study, Acar *et al.* find that bad cryptographic library usability contributes to misuse and, therefore, insecure code [ABF+17]. Krueger *et al.* developed and built upon a wizard to create secure code snippets for cryptographic use cases [KNR+17; KSA+21]. Unlike these prior studies that investigate users of cryptographic libraries, we study the developers of cryptographic libraries, their threat models, and decisions as they relate to *timing attacks*.

Haney *et al.* investigate the mindsets of those who develop cryptographic software, finding that company culture and security mindsets influence each other positively, but also that some cryptographic product developers do not adhere to software engineering best practices (*e.g.,* they write their own cryptographic code) [HTAP18]. We expand on this research by surveying open-source cryptographic library developers concerning their decisions and threat models relating to side-channel attacks.

In the setting of constant-time programming, Cauligi *et al.* [CSJ+19a] carry a study with over 100 UCSD students to understand the benefits of FaCT, a domain-specific framework that enforces constant-time at compile-time, with respect to constant-time programming in C. They find that tool support for constant-time programming is helpful. We expand on their study by surveying open-source cryptographic library developers and considering a large set of tools.

Very recently, there have been calls to make formal verification accessible to developers: Reid *et al.* suggest *"meeting developers where they are"* and integrating formal verification functionality in tools and workflows that developers are already using [RCF+20]. To our knowledge, ours is the first survey empirically assessing cryptographic library developers' experiences with formal verification tools.

8.3. Methodology

In this section, we provide details on the procedure and structure of the survey we conducted with 44 developers of popular cryptographic libraries and primitives. We describe the coding process for qualitative data and the approach for statistical analyses for quantitative results. We explain our data collection and ethical considerations and discuss the limitations of this work.

> **Takeaway:** Multiple factors have been considered to build the survey, code and analyze data, and avoid bias from our experience. The methodology was inspired by previous human factor research and guided by an experienced human factor researcher.

8.3.1. Study Procedure

We asked 201 representatives of popular cryptographic libraries or primitives to participate in our survey. The recruited developers reside in different time zones, and each may have different time constraints. As we were mainly interested in qualitative insights, based on the small number of qualifying individuals and our past experiences with low opt-in rates when attempting to recruit high-level open source developers into interview studies, we opted for a survey with free-text answers.

Questionnaire Development. We used our research questions as the basis for our questionnaire development. However, we also let our experience with the development of cryptographic libraries, constant-time verification tools (both as authors and users), and conducting developer surveys influence the design. Our group of authors consists of one human factors researcher and experts from cryptographic engineering, side-channel attacks, and constant-time tool developers. The human factors researcher introduced and facilitated the use of human factors research methodology to answer experts' research questions posited in this chapter. In particular, the human factor researcher explained methods when appropriate, facilitated many discussions, and helped the team to develop the survey, pilot it, gather feedback and evaluate the results. While iterating over the questionnaire, we also collected feedback and input from members of the cryptographic library development community.

Pre-Testing. Following the principle of cognitive interviews [LSR+22], we walked through the survey with three participants who belonged to our targeted population and updated, expanded, and clarified the survey accordingly.

Recruitment and Inclusion Criteria. We created a list of the most active contributors to libraries that implement cryptographic code, including those that implement cryptographic primitives. If a library had any formal committee for making technical decisions, we invited its members. The list of most active developers was extracted from source control by taking the developers with the largest amount of commits down to a cut-off point that was adjusted per library. Table 8.2 gives an overview of projects for which we invited participants. All authors then identified those contributors that belonged to their own personal or professional networks and invited those in a personalized email. All others were invited by a co-author who is active in the formal verification and cryptography community, for whom we assumed that they would be widely known and have the best chance of eliciting responses. All contributors were sent an invitation with a personalized link. We did not offer participants compensation but offered them links to all the tools we mentioned in our survey and the option to be informed about our results.

8.3.2. Survey Structure

The survey consisted of six sections (see Figure 8.2) detailed below. The full questionnaire can be found in Appendix C.

1. Participant background: We asked participants about their cryptography background, years of experience developing cryptographic code, and experience as a cryptographic library/primitive developer.

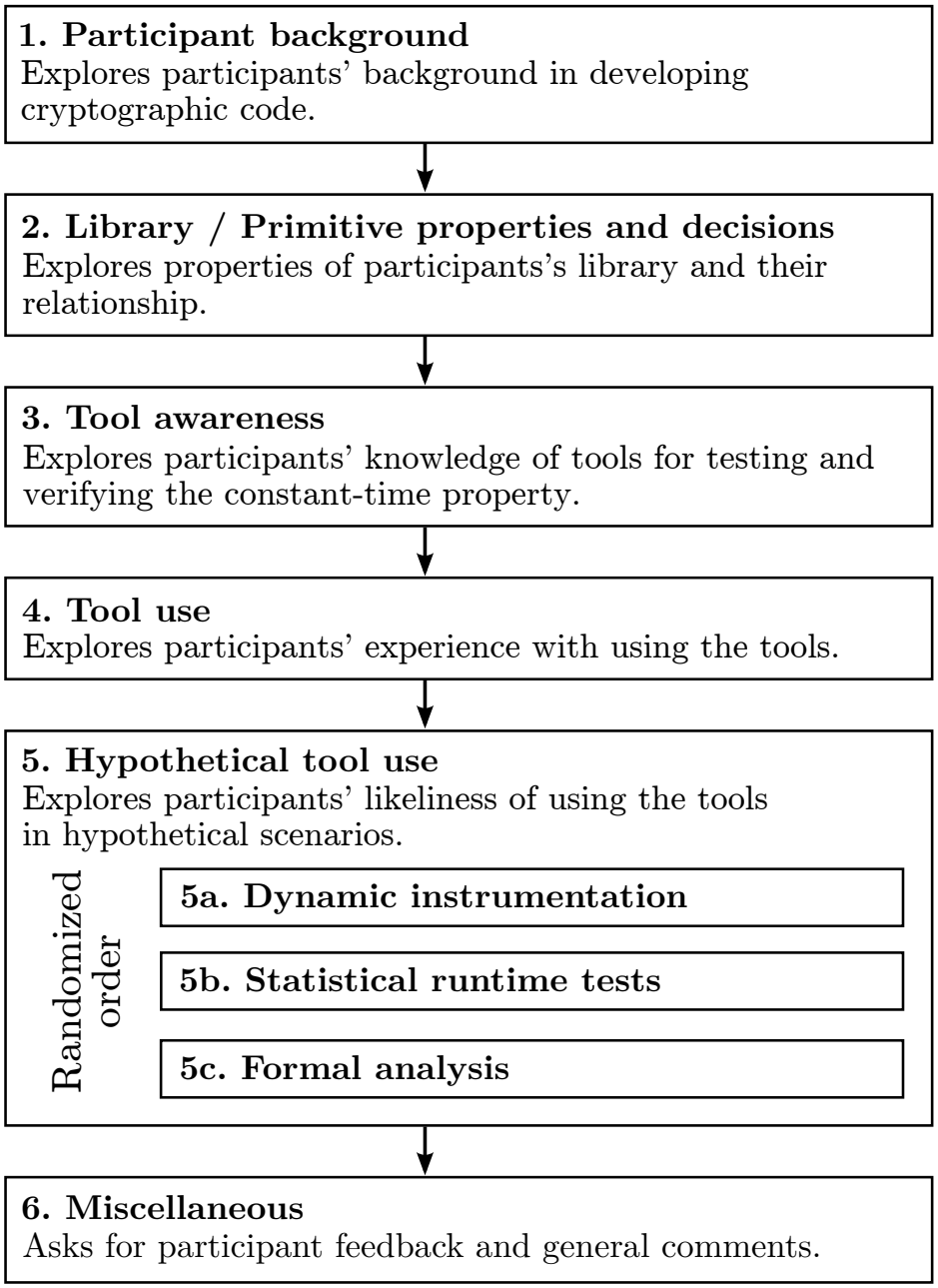


Figure 8.2. – Survey flow as shown to participants.

2. Library properties and decisions: We asked about participants' role in <library>'s development and how they are involved in design decisions for <library>. We asked about the intended use cases for <library>, <library>'s threat model concerning side-channel attacks, whether they consider timing attacks a relevant threat for the intended use of <library> and its threat model, and asked for an explanation for their reasoning. We also asked whether and how <library> protects against timing attacks and whether, how, and how often they test or verify resistance to timing attacks.

3. Tool awareness: We asked whether they were aware of tools that can test or verify resistance to timing attacks. We then listed 25 tools from Section 8.2.2 and asked them whether they were aware of them and how they learned about them.

4. Tool use: We asked participants about their past experience, interactions, comprehension, and satisfaction with using tools to test/verify resistance to timing attacks, including challenges with using them.

5. Hypothetical tool use: We showed participants a description of properties their code would have to fulfill to be able to use a group of tools and a description of the guarantees the tools would give them, asking them about usage intentions and reasoning. The tools were grouped into dynamic instrumentation-based, statistical test-based, and formal analysis tools.

6. Miscellaneous: Finally, we asked about any comments on (resistance to) timing attacks, our survey, and whether they wanted us to inform them about our results.

8.3.3. Coding and Analysis

Those who engaged with participant responses came from different backgrounds, with different views, contributing to the multi-faceted evaluation. Three researchers familiar with constant-time verification and open-source cryptographic library development conducted the qualitative coding process, facilitated by one researcher with experience with human factors research with developers. We followed the process for thematic analysis [BC06]. The three coders familiarized themselves with all free-text answers and annotated them. Based on these annotations, themes were developed, as well as a codebook. The codebook was developed inductively based on questions, and iteratively changed based on responses we extracted from the free-text answers; all codes were operationalized based on discussions within the team. The three coders then coded all responses with the codebook, iterating over the codebook until they could make unanimous decisions. The codebook codifies answers to free-text questions and identifies misconceptions, concerns, and wishes. In some cases where documentation was available, and participant answers were incomplete or ambiguous, or when participants linked to documentation, coders supplemented their code assignment based on the documentation. Our coding process was only one step in the quest for our goal: identifying themes and answering our research questions. All codes were discussed and eventually agreed upon by three coders.[3] In line with contemporary human factors research, we omit inter-coder agreement calculations [MSF19]. For the comparison of the likelihood of using certain tools with specific requirements in exchange for guarantees (Q5.1, Q6.1, Q7.1), we used Friedman's test with Durbin post-hoc tests [Con98] with Benjamini-Hochberg multiple testing corrections [BH95].

[3]Our codebook is available at https://crocs.fi.muni.cz/public/papers/usablect_sp22.

8.3.4. Data Collection and Ethics

While our survey was sent to open-source contributors without solicitation, we only emailed them up to twice. During and after the survey, they could opt out of participation. We do not link participant names to results nor participant demographics to libraries to keep responses as confidential as possible. We also do not link quotes to libraries or their developers and report mostly aggregate data. Quotes are pseudonymized. Our study protocol and consent form were approved by our institution's data protection officer and ethics board and determined to be minimal risk. Participant names and email addresses were stored separately from study data and only used for contacting participants.

8.3.5. Limitations

Like all surveys, our research suffers from multiple biases, including opt-in bias and self-reporting bias. However, we were pleasantly surprised that for 27 out of the 36 libraries we selected, we received at least one valid response. Participants may over-report desirable traits (like caring about side-channel attacks or protecting against them), and underplaying negative traits (like making decisions ad-hoc). However, their reporting generally tracked with official documents and our a priori knowledge about the libraries. The projects represent a selection and are not representative of all cryptographic libraries. However, we took great care in inviting participants corresponding to a variety of prominent, widely used libraries as well as smaller but popular libraries and primitives, as assessed by multiple authors who work in this space.

8.3.6. Data cleaning & Presentation

We emailed 201 ♀[4] listed as most active contributors to 36 libraries/primitives, finding alternate emails for those emails that bounced. 2 ♀ emailed us to tell us that they did not think they could contribute meaningfully. In total, 71 ♀ started the survey. We removed all 25 incomplete responses. We removed two participants because they gave responses about a project of their own, instead of the library we asked them about. From here, we report results only for the 44 valid participants. For statistical testing and figures for hypothetical tool use, we report results for the 36 participants who gave answers for all three tool groups. We merged participants' answers about using a ctgrind-like approach but without the use of ctgrind itself (as it is no longer necessary as Valgrind can directly do this) into the ctgrind tool answers.

Participants spent an average of 32 minutes on the survey, and left rich free text comments. We generally received positive feedback and high interest in our work, and 35 ♀ asked to be sent our results, with 33 ♀ agreeing to be contacted for follow-up questions. Whenever we report results at the library level, we merge qualitative answers from all participants corresponding to that library. Whenever the answers are additive, we add them together without reporting a conflict (*e.g.,* when one developer tests a library in one way while another tests it a different way, we report both). When the answers are claims of a level (*e.g.,* resistance to timing attacks), we report the highest claim. Otherwise, whenever we encounter conflicting opinions, we report on this conflict.

[4]From now on, we use the ♀ symbol to denote the participants.

8.4. Results

> **Takeaway:** We identified a "leaky pipeline" of why constant-timeness is not a solved problem in cryptographic software (see Figure 8.1). Starting from complete awareness of the potential of timing attacks, we identified multiple points where varying levels of support could better support the widespread use of constant-time cryptographic code. For example, availability, maintenance, and usability play major roles for developers to not adopt tools into their development process, as do tradeoffs, and assumed high cognitive and time effort investments.

In this section, we answer our research questions based on our survey results. Between full awareness and low levels of protection against timing attacks, we identify reasons for (not) choosing to develop and verify constant-time code, including a lack of (easy-to-use) tooling, tradeoffs with competing tasks, understandable concerns, and misconceptions about current tooling. We identify that participants would generally like the guarantees offered by tools but fear negative experiences, code annotations, and problems with scalability.

8.4.1. Survey Participants

Library developers

We successfully recruit experienced cryptographic library developers, including the most active contributors and decision-makers. We ended up with 44 ♀ recruited via direct invitation. Of our participants, 4 ♀ were the only developer in their project, 9 ♀ were project leads, 11 ♀ were core developers, 19 ♀ were maintainers, 11 ♀ were committers, 3 ♀ were contributors without commit rights. These classes are non-exclusive self-reports. 40 ♀ said they were involved in the library decision processes, while only 4 ♀ were not involved.

Participants had strong backgrounds in cryptographic development, reporting a median of 10 years of experience (sd = 7.75), and qualitatively reporting strong engagement with various projects, for example reporting involvement in security certifications:

> *"I've worked on open source and closed source cryptography libraries, dealt with various Common Criteria EAL4+ products"* (P1)

As for the participants' concrete background in cryptography, 17 ♀ reported an academic background, 15 ♀ took some classes on cryptography, 32 ♀ had on the job experience, 6 ♀ teach cryptography, for 15 ♀ cryptography is (also) a hobby, 27 ♀ have industry experience in cryptography.

Libraries

We ended up with participants from 27 prominent libraries, such as OpenSSL, BoringSSL, mbedTLS or libgcrypt. Participants gave or linked to descriptions of a broad range of use cases for cryptographic libraries. As intended platforms, 23 gave servers, 22 desktop, 14 embedded device (with OS, 32 bit), 4 mobile, and 1 micro-controller (no OS, 8/16 bit). For targets, 7 stated TLS, 12 protocols, 2 services, 1 cloud, 2 operating systems, 1 crypto-currency, and 2 corporate internal purposes. Libraries had varying decision-making processes: 9 made

decisions by discussion, 2 by voting, 3 by consensus, and for 11, decisions were made by the project leads who had a final say.

8.4.2. Answering Research Questions

Threat models (RQ1a)

Here, we answer the research question of whether timing attacks are part of library developers' threat models (RQ1a). We found that *all participants were aware of timing attacks.* Generally, when a threat model is defined for a cryptographic library, it mostly includes timing attacks. However, strict and absolute adherence to constant-time code is most often not required. In practice, developers tend to distinguish vulnerabilities that are "easy" to exploit (*e.g.,* remote timing attacks) from the others (*e.g.,* locally exploitable attacks). When asked specifically about the library's threat models with respect to side-channel attacks, 20 libraries claimed remote attackers are in their threat model, 16 included local attackers, 1 included speculative execution attacks, 2 included physical attacks and 2 included fault attacks. Some libraries expressed that they consider some classes of attacks in their threat model if they are easy to mitigate, 2 would do so for local attacks and 1 for physical attacks. The general attitude towards side-channel attacks varied, 2 said that all side-channel attacks are outside their threat model and 10 said that their protections against side-channel attacks are best effort. For example, one participant said:

> *"Best-effort constant-time implementations. CPU additions and multiplications are assumed to be constant-time (platforms such as Cortex M3 are not officially supported)."* (P2)

Another one implied a progressive widening of their threat model regarding timing attack in their statement:

> *"Protections against remote attacks, and slow movement to address local side channels, though the surface is wide."* (P3)

In a follow-up question, 23 libraries agreed that timing attacks were considered a relevant threat for the intended use of the library and its threat model, while this was not true for 2 libraries. We did not get this information for 2 libraries.

Reasons for considering timing attacks as relevant for their threat model were given as the ease of doing so (2), the threats of key-recovery in asymmetric cryptography (3), user demands (1), fear of reputation loss (1), use in a hostile environment (6), that attacks get smarter (1), the (rising) relevance of timing attacks (9), personal expectations (5), a connected environment (2), or the large scope of the library/of timing attacks (3).

Reasons for not considering timing attacks as part of their threat model were stated as this not being a goal of the library (2) or that they only consider more "practical" attacks.

Resistance against timing attacks (RQ1b)

Here, we answer the research question of whether libraries claim resistance against timing attacks (RQ1b). Many libraries do not have a systematic approach to address timing attacks; they only consider fixing "serious" vulnerabilities that could be exploited in practice. This might result in vulnerable code that can be exploited later with better techniques for recovering leaking information. We also encountered differing answers from different

participants regarding the suitedness of random delays as mitigation. Out of the 27 total libraries, 13 claimed resistance against timing attacks. An additional 10 claimed partial resistance, 3 claimed no resistance, and for 1, we obtained no information.

We also asked how the development team decided to protect against timing attacks. For 4 libraries, participants reported that one person made this decision, for 12 it was a team decision, for 2 it was a corporate decision (where high-level management makes a decision or the team decided locally based on a corporate mission statement), for 14 libraries, participants reported that a priority trade-off caused their decision (*e.g.,* lack of time to fully enact the decision) and 5 inherited the decision from previous projects or developers.

For 6 it was obvious that they needed to protect against timing attacks. For example, one participant stated:

> *"There was no decision, not even a discussion. It was totally obvious for everybody right from the start that protection against timing attacks is necessary."* (P4)

Another one said:

> *"It's just how you write cryptographic code, every other way is the wrong approach (unless in very specific circumstances or if no constant-time algorithm is known)."* (P5)

Another stated

> *"It became clear that these attacks transition from being an "academic interest" to a "real-world problem" on a schedule of their own development. If something is noticed we now tend to favor elimination on first sight without waiting for news of a practical attack."* (P6)

Contrarily, another said:

> *"Basically a tradeoff of criticality of the algorithm vs practicality of countermeasures. Something very widely used (*e.g., *RSA, AES, ECDSA) is worth substantial efforts to protect. Something fairly niche (eg Camellia or SEED block ciphers) is more best-effort"* (P7)

This reasoning of waiting for attacks to justify expending the effort was also reported by another participant:

> *"For many cases there aren't enough real world attacks to justify spending time on preventing timing leaks."* (P8)

Timing attack protections (RQ2a)

Here, we answer the research question of how developers choose to protect against timing attacks (RQ2a). Developers address timing attacks in various ways, for example, by implementing constant-time hacks (*e.g.,* constant selecting), implementing constant-time algorithms of cryptographic primitives, using special hardware instructions (CMOV, AES-NI), scatter-gathering for data access, blinding secret inputs, and slicing. Many are interested and willing to invest effort into this - to various degrees, as P9 puts it:

> *"[T]hey're not that hard to mitigate, at least with the compilers I'm using right now"* (P9)

Others are deterred by the lack of (easy-to-use) tooling.

We asked developers of the 23 libraries who considered timing attacks at least partially if and how their library protects against timing attacks.

For 2 libraries, participants reported that they use hardware features (instead of leak-prone algorithms) that protect from timing attacks such as AES-NI. For example, P7 said:

> *"AES uses either hardware support, Mike Hamburg's vector permute trick, or else a bytesliced version."* (P7)

For 21 libraries, participants said that they use constant-time code practices, which should, in theory, mean that code is constant-time by construction but may be vulnerable to timing attacks after compilation. For example, P2 explained that:

> *"Conditional branches and lookups are avoided on secrets. Assembly code and common tricks are used to prevent compiler optimizations."* (P2)

For 9 libraries, participants explained that they choose known-to-be constant-time algorithms, but may suffer from miscompilation issues and end up non-constant-time. As an example, P7 said:

> *"If I know of a "natively" const time algorithm I use it (eg DJB's safegcd for gcd)."* (P7)

For 7 libraries, participants said they use "blinding", which means using randomization to "blind" inputs on which computation is performed, thereby destroying the usefulness of the leak. As P7 said:

> *"If blinding is possible (eg in RSA or EC scalar mult) it is used, even if the algorithm is otherwise believed constant-time. Since sometimes compilers do not behave as you expect, maybe I've made a mistake and thinking something is constant-time but it isn't, and they may provide some defense against power analysis. "* (P7)

For 2 libraries, participants said that they protect through bitslicing, *i.e.,* the implementation uses parallelization on parts of the secrets, hiding leaks. As one participant described:

> *"For instance, the constant-time portable AES implementations use bitslicing."* (P11)

For 2 libraries, participants reported protecting by "assembly", *i.e.,* they have a specialized low-level implementation for protecting against compilers doing non-constant-time transformations. One participant noted the prohibitive cost of this practice, explaining:

> *"We do not write all constant-time code in assembly because of the cost of carrying assembly code. It is possible that the compiler may break the constant-time property. We spot-check that using Valgrind."* (P12)

For 1 library, timing leaks are made harder to detect by adding random delays.

Most developers focus on asymmetric crypto. Some do not consider old primitives, such as DES, which is still used in payment systems as Triple-DES. For 5 libraries, participants stated that they only protect a choice of modules: those libraries have multiple implementations, of which only some might be constant-time, maybe even insecure by default. A participant also mentions bignum libraries being specifically hard to secure.

> *"Legacy algorithms like RC4 and DES are out of scope. If you use the <libraries'> "BIGNUM" APIs to build custom constructions, it's probably leaky, since bignum width management is complex."* (P13)

This claim is supported by academic literature as well: *"lazy resizing of Bignumbers in OpenSSL and LibreSSL yields a highly accurate and easily exploitable side channel"* [WSBS20].

For 1 library, protection against timing attacks was reported to be still in progress, *e.g.,* they try to use constant-time coding practices throughout the library, but this is still in development due to the large legacy code base. Two answers from participants of libraries illustrate very different phases of solving this problem

> *"All decisions in a side project are limited by the available resources. There's a report about a new attack which proposes a new counter-measure: Does someone have the time to implement it? Yes - cool, let's do it. No - fine, let's put it on the ToDo list."* (P15)

> *"Very early on in its development these guarantees were much weaker, and in a few cases, approaches were used that turned out to be known to be imperfect."* (P16)

Testing of timing attack resistance (RQ2b, RQ2c)

In Software Engineering, testing code for the properties it should achieve is commonplace and generally considered best practice [Ber07]. We, therefore, were interested in the practice of testing and verification for constant-time also. Here, we answer the research questions about whether, how, and how often libraries test for/verify resistance against timing attacks (RQ2b, RQ2c).

For 21 libraries, at least some type of testing was done, of which 14 were fully, and 7 were partially tested. 6 were not tested including the 2 libraries which claimed timing attacks are not relevant. 24 🎗 personally tested their libraries.

Of those, 12 stated they have tested manually, and 11 stated they tested automatically. Those two answers are not exclusive since 7 libraries that test code automatically have also been tested manually. For manual testing, 6 libraries analyzed (parts of) their source code, 4 libraries analyzed (parts of) their binary, 5 did manual statistical runtime testing for leakage, and 1 ran the code and looked at execution paths, debugging as it ran. the following quote conveys the experience quite graphically.

> *"Originally, me, a glass of bourbon, and gdb were a good trio. But that got old pretty quick. (The manual analysis part – not the whiskey.)"* (P17)

For those who did automated testing, 9 libraries used a Valgrind-based approach, 2 used ctgrind, 1 used Memsan, 1 used TriggerFlow, 1 used DATA, and 1 reported automated statistical testing without specifying further.

For the participants who did at least partial testing for resistance to timing attacks, we asked for testing frequency. For 1, the testing was only done once. For 11, participants reported manual or occasional testing. For 4, participants reported testing on release. For 6 libraries, participants reported that testing for resistance to timing attacks was part of their continuous integration. For 11 libraries, we did not obtain information on testing or testing frequency. These varying answers suggest that despite a common awareness of timing attacks, cryptographic developers never came to a consensus on the best way to address timing attacks in practice.

Tool awareness (RQ3a)

In order to effectively test, developers should be able to leverage existing tooling created for testing and/or verifying that source code, or compiled code, runs in constant time. Here, we answer the research question of whether participants are aware of the existence of such tooling (RQ3a). We asked participants whether they were aware of tools that can test or verify resistance against timing attacks, also showing them a list of tools from Table 8.1. We asked them whether they had heard about any of those tools with regards to verifying resistance against timing attacks. Table 8.3 shows the individual tool awareness and use numbers results. In particular, it shows 33 🎗 being aware of at least one tool and 11 🎗 being unaware of any tool. ctgrind was most popular (27 🎗 heard of it; 17 🎗 had tried to use it), followed by ct-verif (17 🎗 heard of it; only 3 🎗 tried to use it) and MemSan (8 🎗 heard of it; 4 🎗 tried to use it). DATA had been used by 2 🎗, all others by no more than 1 🎗.

Table 8.3. – Tool awareness and use

Tool	Aware	%	Tried to use	%
ctgrind [Lan10]	27	61.4%	17	38.6%
ct-verif [BGA+16]	17	38.6%	3	6.8%
MemSan [Tea20]	8	18.2%	4	9.1%
dudect [RBV17]	8	18.2%	1	2.3%
timecop [Nei]	8	18.2%	1	2.3%
ct-fuzz [HEC20]	7	15.9%	1	2.3%
CacheD [WWL+17]	6	13.6%	1	2.3%
FaCT [CSJ+19a]	6	13.6%	0	0.0%
CacheAudit [DFK+13]	5	11.4%	0	0.0%
FlowTracker [RPA16]	4	9.1%	1	2.3%
SideTrail [ACE+18]	3	6.8%	0	0.0%
tis-ct [Cuo]	3	6.8%	0	0.0%
DATA [WZS+18; WSBS20]	2	4.5%	2	4.5%
Blazer [AGH+17]	2	4.5%	0	0.0%
BPT17 [BPT17]	2	4.5%	0	0.0%
CT-WASM [WRP+19]	2	4.5%	0	0.0%
MicroWalk [WMES18b]	2	4.5%	0	0.0%
SC-Eliminator [WGSW18]	2	4.5%	0	0.0%
Binsec/Rel [DBR20]	1	2.3%	0	0.0%
COCO-CHANNEL [BSBP18]	1	2.3%	0	0.0%
haybale-pitchfork [UCSa]	1	2.3%	0	0.0%
KMO12 [KMO12]	1	2.3%	0	0.0%
Themis [CFD17]	1	2.3%	0	0.0%
VirtualCert [BBC+14]	1	2.3%	0	0.0%
ABPV13 [ABPV13]	0	0.0%	0	0.0%
None	11	25.0%	25	56.8%

For those tools they had heard about, we asked them where they had heard about them. Overall, participants were recommended a tool by a colleague 33 times, heard of a tool

from its authors 20 times, read the paper of the tool 27 times, read about the tool in a different paper or blog post 42 times and heard of it some other way 24 times. 2 🧍 were involved in a development of a tool. A general tool they are already using can also be used for constant-time-analysis, which P18 learned through our survey:

> *"I already use MemSan primarily for memory fault detection. Was not aware of its use for side-channel detection but will try it in future since it is already integrated with my workflow to some extent"* (P18)

Again for the tools they were aware of, we asked which (if any) they had (tried to) use for verifying or testing resistance to timing attacks. Table 8.3 displays the results, with 19 🧍 having tried to use at least one tool and 25 🧍 having never tried any of the tools.

Tool experience and use cases (RQ3b)

Here, we answer the research question about which experiences participants made with tools (RQ3b). As we were anecdotally aware that tools may be hard to obtain, unmaintained, and may be closer to research artifacts than ready-to-use tooling, we were interested in participants' experiences, finding that experience varied by tool, use cases, and expectations. We, therefore, asked participants to describe the process of using the tools.

12 🧍 reported that they managed to get the respective tool to work at least once, but not necessarily repeatedly. In contrast, 3 🧍 reported that the tool they attempted to use failed to work even once, for various reasons, including excessive use of resources, such as effort, time, Random Access Memory (RAM), CPU cores, machines, ... One participant said of the DATA tool:

> *"it uses a ridiculous amount of resources"* (P17)

2 🧍 reported that they had integrated the tool into CI and were using it automatically. 12 🧍 reported that they used it manually, of which 6 🧍 said they use it during development, and 6 🧍 said they use it after development, on release. A participant said:

> *"Periodically, and manually, used when altering / writing code to check constant-time property."* (P12)

For those who had heard of specific tools but had not attempted to use them, we were also interested in their reasoning. The reasons were varied, many including a lack of resources such as time (26 🧍) or RAM, CPU cores and machine (1 🧍). Participants also reported on bad availability (4 🧍), and maintenance (5 🧍), as well as insufficient language support (4 🧍), and other usability issues, such as problems with setting up the tool (3 🧍), or getting it to work properly post setup (1 🧍). The difficulty or impossibility of fulfilling the required code changes, such as markup for secret/public values, memory regions/aliasing, and additional header files was also a problem (reported by 1 🧍), as was the inability to ignore reported issues, once flagged by the tool (8 🧍).

Some reported not needing the respective tool (22 🧍), using other tools (18 🧍), gave reasoning that to our understanding was based on misconceptions of the respective tool (2 🧍), or reported having been unaware of the tool's capabilities in the context of resistance to timing attacks (1 🧍).

One participant also said the tool was used to verify a security disclosure.

"Tried to use to reproduce results, verify disclosures. Tried to use it to discover new defects in existing code." (P14)

— since the tool is later stated as in use by another member of the same project, this confirms that the tool not only verified the initial defect, but works as planned.

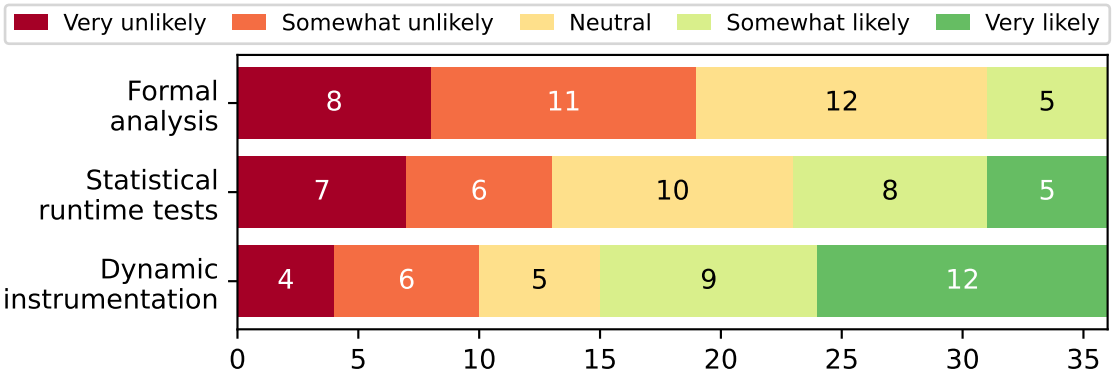


Figure 8.3. – Reported likeliness of tool use based on requirements and guarantees.

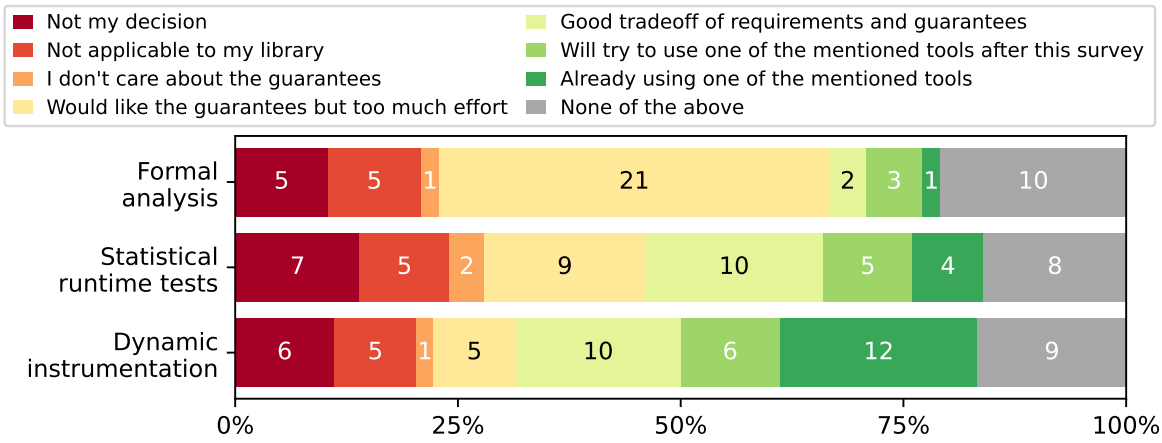


Figure 8.4. – Participant reasoning behind their likelihood of tool use.

Potential Tool use (RQ4)

In addition to understanding participants' current threat models and behaviors concerning constant-time code, we were also interested in what they thought about potential future use of testing/verification tools, and whether they would potentially be willing to fulfill specific requirements in exchange for guarantees (RQ4, see Figure 8.3). Generally, they were most willing to use dynamic instrumentation tools and spoke about them the most positively, whereas they mainly mentioned drawbacks when asked about formal analysis tools.

We presented the participants with the requirements and guarantees offered by three categories of tools: dynamic instrumentation-based tools, statistical runtime tests, and formal

analysis tools.[5] We then asked them to rate their likeliness of using the presented group of tools on a 5-point Likert scale from "1=very unlikely" to "5=very likely". Figure 8.3 shows a strong preference for dynamic tools, while formal analysis tools are least likely to be used. We perform statistical tests on these ratings to establish that these differences are statistically significant. We find a significant difference in participants' self-reported likeliness to use tools in the different categories (Friedman Test-Statistic=18.477, $p<0.0001$). Post-hoc testing showed that participants are significantly more likely to use dynamic instrumentation based tools like ctgrind (mean=3.53, sd=1.38) than statistical tools (mean=2.94, sd=1.31; $p=0.023$, Durbin-post-hoc (DPH), Benjamini-Hochberg-corrected (BH)) and formal analysis tools (mean=2.38, sd=0.98; $p<0.0001$, DPH, BH-corrected). The difference between statistical and formal analysis tools was not significant ($p=0.18$, DPH, BH-corrected). Specifically, while 21 🧍 reported being somewhat likely or very likely to use a dynamic testing tool for resistance to timing attacks in the future, 13 🧍 reported the same for statistical runtime test tools, and only 5 🧍 said they were somewhat likely to use formal analysis tools.

We also asked participants to clarify their reasoning by choosing explanations (see Figure 8.4). Results show that participants would like the guarantees formal analysis tools provide, but perceive them as requiring too much effort (21 🧍) compared to the other tools (9 🧍 statistical runtime tools, 5 🧍 dynamic instrumentation tools). On the other hand, more participants think that the tradeoff of effort and guarantees is acceptable for dynamic (10 🧍) and statistical tools (10 🧍) than formal analysis tools (2 🧍). More details on participants' reasoning follow.

Dynamic Instrumentation Tools. For dynamic instrumentation tools, some participants were happy with the limited guarantees, understanding the tradeoff clearly.

> *"We currently use MemSan and Valgrind because they have very low maintenance since they pretty much come with the operating system, and we could get useful results from them with a few days' work. We are aware of their limitations (they miss non-constant-time parts, and of course they can only test code in the conditions where it is executed as part of the tests)."* (P19)

The approach taken by tools like ctgrind is understandable to developers, so much so that some came up with it independently:

> *"We independently came up with this approach and were using it [before we] knew ctgrind existed."* (P9)

One participant specifically commented on the effort required to create and maintain annotations:

> *"A thing this survey might be underestimating is also the cost of code annotations: it's not just about having someone annotating the code properly (which already is quite a lot of effort) but there might be resistance for inclusion of such annotations in the code base as they add a maintenance burden for the project. Maintainers should fully understand the notation syntax and get proficient in it to spot instances where annotations need to be updated, moved, etc."* (P20)

Statistical Tools. For tools based on statistical tests of the runtime, 7 🧍 expressed that the guarantees provided by the tools are limited. One participant explained:

[5] For the survey questions see the Appendix sections C.4,C.5 and C.6.

> *"I am dubious that it would provide much value over existing mechanisms. Also, CI currently runs on shared hosts which are timing noisy. From this noise I would expect [...] false positives [...]"* (P21)

Another participant also expressed concern over the guarantees and false positives/negatives:

> *"The requirements seem straightforward, but a statistical test seems likely to cause both false negatives and false positives."* (P13)

Formal Analysis Tools. Participants had strong feelings about the lack of usability of formal analysis tools:

> *"I'm very interested in these sorts of tools, but so far it seems formal analysis tools (at least where we've tried to apply it to correctness) are not really usable by mere mortals yet. I would be happy to be proven wrong, however!"* (P13)

The fact that compiler optimizations can introduce timing leaks that will not be detected by tools working at the source code level was highlighted by a few participants:

> *"Static analysis on the source code in most programming languages is NOT sound: it misses compiler optimizations that introduce secret-dependent flows."* (P19)

and another one explaining

> *"I'm much more worried about compilers failing to preserve constant-time code, ..."* (P13)

While 4 ⚷ mentioned their expectation of a higher effort to create the necessary markup for formal analysis-based tools, expectations of the scalability of these tools seem to be in line with other categories of tools.

Additionally, we found that participants were intimidated by the theory-heavy approach by formal analysis-based tools, thinking of formal verification in general.

> *"I have no experience with formal verification toolchains"* (P23)

More academically focused formal analysis tools also suffer from a maintenance problem if the developers have moved on to other research:

> *"Who knows if the toolchain is still maintained in a year?"* (P5)

In conclusion, dynamic tools are mostly criticized for requiring code annotation, while statistics ones are viewed critically because of their poor guarantees. However, participants were most critical towards formal analysis tools. Some doubt that such tools would be maintained or question the fact that they would provide large support for different platforms. While these drawbacks are real, they do concern all tools, but participants point them out mainly for formal analysis tools. Some participants mention that such tools have steep learning curves, as they are not only unfriendly to use but also require specific knowledge. We notice that developers using ct-grind took their time to explain how it works (they were never asked to), while participants remained vague about formal analysis tools. Only one participant actually uses such a tool, and a few have tried but did not succeed. However, many qualify such tools as uneasy to use, inefficient, lacking wide support, unable to verify external code, based only on the code, hard to be CI automated, adding very little confidence,

and possibly unmaintained in the future.

Misconceptions

Despite surveying an expert population of cryptographic library developers, our study pointed out some misconceptions and differences of opinion about constant-timeness, timing attacks, and verification/testing tools. Those may deter from analysis tool use, and may contribute to more hidden timing vulnerabilities, ultimately making it harder to solve the timing attack problem in practice.

Some participants seemed to think constant time is easily achieved. This logic implies that if a project has a timing vulnerability, they have made a basic mistake.

> *"Writing constant-time code, contrary to writing [...] memory-safe code, is not hard, if you do it explicitly from start (caveat: when there's Gaussian rejection sampling in a lattice system, it _is_ hard[...]"* (P11)

This ties in with code annotation not being usable when the secretness of variables changes, specifically as mentioned with rejection sampling. This misconception is based on the most common use of annotations. If the annotations allow for declassification of variables, this problem can be resolved granularly. Not all tools allow this, though, so the misconception that this is true for all tools may have taken root.

One participant suggested they do not need to test code if they correctly write constant-time code:

> *"In that sense, the guarantees offered by these tools are not worth putting effort into running them, at least in the case of <library>, where all code was designed to be constant-time"* (P11)

This sentiment comes with several problems: on the one hand, humans make mistakes, so testing code is a best practice in software engineering for precisely this reason. Additionally, compiling code that does fulfill the constant-time property may create problems, as the compiler may change the original control flow while adding some optimizations.

While talking about compilation units and control-flow, a partial misconception can be found in verification scope:

> *"a lot of code will exist outside of the boundaries of the library. A project using <library> would be more likely to be successful."* (P20)

While the library may not know may not know which inputs are secret, looking at an API should make it clear which inputs can be secret, and the constant-time criterion could be tested for all of them without knowing the actual usage patterns.

Furthermore, the different answers about random delays and statistical analysis tools show that there is no universal consensus among the participants. A participant said:

> *"Anything involving secret data, and in particular private-key data, has the timing dithered and with throttling of repeated attempts to make attacks of this kind difficult."* (P24)

We are skeptical about this due to the results of Brumley and Tuveri [BT11]. If a side channel signal is measured as a timing difference between executions, adding a random noise distribution to these executions will reproduce a similar difference if enough samples of the executions are obtained. This can be done in parallel from different sources or, over a long

time, going around the throttling defense. A more practical quote is from P9:*"We once tried to test actual execution timings, but it wasn't reliable. We no longer do that. Now we use Valgrind."*

Lastly, even if cryptography is rather heavy in mathematics, some participants associate math/formal analysis as a barrier to using tools from that research area.

> *"[P]roving things like loop bounds is often arcane. Also, it's knowledge that would present a barrier to new engineers joining the team."* (P12)

This is most likely a misconception, potentially caused by unclear writing in formal analysis tools' documentation, or scientific publications that do not separate tool use from general formal verification and theorem proving.

Developer Concerns and Wishlist (RQ5)

In addition to misconceptions, participants also voiced understandable concerns about constant-time development and wished for verification tools that would allow them to use these tools more effectively (RQ5). Major concerns were voiced about the tools' resource usage being too high (see section 8.4.2).

In addition to these issues, P14 listed concerns as: *"the execution time of static and dynamic analyzers tailored for SCA, the need for human interaction, the rate of false positives, etc. are usually preventing a systematic adoption"*. The issue with flagging false positives and not linking false positives and negatives was addressed by another participant also:

> *"We noticed a couple false positive, where there *is* a path from the contents of the buffer to timings, but we decided that doesn't leak any meaningful secret."* (P9)

They also mentioned security concerns for tools based only on the source code. These may miss vulnerabilities due to miscompilation, as explained by P13: *"Any "constant-time" code is an endless arms race against the compiler"*.

Interestingly, participants had many precise ideas for what could be done to improve the status quo of testing/verification tools. For example, for better usability, they ask for the ability to ignore some issues and/or some part of the code, as noted by P14: *"Also, expect a lot of "noise" from BIGNUM behavior that is not CT and requires a full redesign to be fixed."*

We saw many wishes for improvements concerning annotations, asking for external annotations. Participants also asked for easy maintenance of code annotations (see 8.4.2), and requested that tools work on complex code, as P14 explained: *"even for expert users the chances of exposing something non-consttime to remote attackers are high, especially given the complex nature of <library> under the hood."* They also asked for test cases to be fast to set up to avoid a

> *"non-trivial amount of effort to set up comprehensive tests."* (P14)

To address the issue of scale, they want to be able to use tools in CI. Otherwise, when the code changes, the guarantees are lost. This means that error code outputs, easy CI setup, and runtime are essential, as explained by P19: *"Static analysis tools tend to have a high engineering overhead: getting the tool to run, deploying it to CI systems, maintaining the installation over the years."* Similarly, participants demanded that tools not require rewrites of their code: P2 ruled out an *"awesome tool"*, because it *"cannot verify existing code."* Participants also required no restricted language or environment for their code instead of

"a pretty special-purpose language" (P26)

. Similarly, they asked for no use of a specialized compiler; as P4 stated: *"Requiring a dedicated compiler sounds like a potential problem."* Generally, they asked for integration into the type system and APIs they are already using, so the project already has a form of security annotations for the users of their API, which a tool should be able to integrate for its analysis:

> *"which values are public and which private, we have flags on APIs to allow the caller to specify this too"* (P28)

They also requested long-term available source code and long-term maintenance. As P25 stated, tools being unavailable or unmaintained makes it impossible to use them.

8.5. Discussion

Based on our findings, we make suggestions for four groups of actors who can take action to make cryptographic code resistant to timing attacks: tool developers, compiler writers, cryptographic library developers, and standardization bodies.

8.5.1. Tool developers

Even though we selected a subset of well-known tools from the wide diversity of available tools, 25% of the developers who answered our survey did not know about any of them. Some developers learned about the tools from our survey. Only 38.6% actually using any of the tools shows that their adoption is limited. This can be partially explained by the relative youth of the tools, as most tools are less than five years old. However, we believe that many other factors come into play: tools may be research prototypes that are difficult to install, not available, or not maintained; they may not be evaluated on popular cryptographic libraries, raising concerns about applicability and scalability; they may be computationally intensive, making their use in CI unlikely; they may not be published in cryptographic engineering venues. In addition to the specific recommendations from the previous section, we recommend the community of tool developers to:

1. make their tools publicly available, easy to install, and well-documented. Ideally, tools should be accompanied by tutorials targeted to cryptographic developers; making a tool easier to install by providing Linux distribution packages lowers the barrier to adoption;

2. publish detailed evaluations on modern open-source libraries, creating or using a common set of benchmarks. Supercop [BL09] is one such established benchmark;

3. focus on efficient analysis of constant-timeness rather than computationally expensive analysis of quantitative properties, which seem to be of lesser interest. Ideally, tools should be fast enough to be used in CI settings;

4. make their tools work on code with inline assembly and generated binaries to be fully usable by all developers;

5. promote their work in venues attended by cryptographic engineers, including CHES, RWC, and HACS.

Ultimately, we recommend tool developers to follow Reid *et al.*'s recent advice to "meet developers where they are" [RCF+20].

8.5.2. Compiler writers

Developers are very concerned that compilers may turn constant-time code into non-constant-time code. To avoid this issue, developers often use (inlined) assembly for writing primitives. This approach guarantees that the compiler will not introduce constant-time violations but may negatively affect portability and makes analysis more complex. In order to make the integration of constant-time analysis smoother in the developer workflow, we recommend compiler writers to:

1. improve mechanisms to carry additional data along the compilation pipeline that may be needed by constant-time verification tools. This would allow cryptographic library developers to tag secrets in source code and use constant-time analysis tools at intermediate or binary levels;

2. support secret types, as used by most constant-time analyses, throughout compilation, and modify compiler passes so that they do not introduce constant-time violations, and prove preservation of the constant-time property for their compilers. This would allow cryptographic library developers to focus on just their source code;

3. more generally, offer security developers more control over the compiler, so code snippets that implement a countermeasure (*e.g.,* replacing branching statements on booleans by conditional moves) are compiled securely.

8.5.3. Cryptographic library developers

Cryptographic library developers are aware of timing attacks, and most consider them part of their threat model. In order to eliminate timing attacks, we recommend library developers:

1. make use of tools that check for information flow from secrets into branch conditions, memory addresses, or variable-time arithmetic. Ideally, the use of such tools is integrated into regular continuous-integration testing; if this is too costly, a systematic application of such tools for every release of the library may be a suitable alternative;

2. eliminate all timing leaks even if it is not immediately obvious how to exploit them. Attacks only get better and many examples of devastating timing attacks in the past exploited *known* leakages with just slightly more sophisticated attacks techniques;

3. state clearly which API functions inputs are considered public or secret. With a suitable type system, such information becomes part of the input types, but as long as mainstream programming languages do not support such a distinction in the type system, this information needs to be consistently documented. Doing so makes it easier to *use tools* for automated analysis and harder for programmers to *misuse* library functions due to misunderstandings about which inputs are actually protected.

8.5.4. Standardization bodies

A recent paper [BBB+21] advocates for the importance of adopting tools in cryptographic competitions, standardization processes, and certifications. We recommend that submitters are strongly encouraged to use automated tools for analyzing constant-timeness, and that evaluators gradually increase their requirements as constant-time analysis technology matures. Standardization bodies should try to avoid the use of cryptographic algorithms leaking timing information. In the case of Dragonfly, multiple timing attacks have been discovered (as presented in chapter 7) before the Wi-Fi Alliance upgraded the standard to use a deterministic algorithm with a secure design.

Dragonstar: Formally Verified Cryptography for Dragonfly 9

In this chapter, we explore another lead to get side-channel free implementations: high-assurance cryptography. We introduce *Dragonstar*, an implementation of Dragonfly that uses formally verified code for all cryptographic operations, and we demonstrate how it can be embedded within hostap without significantly impacting performance.

In section 9.1, we motivate the need for formally verified implementation and distinguish secret enforcement from code verification. We give an overview of existing solutions and justify our choice to use HACL*. In section 9.2, we explain the basics of formally enforcing secret independence. We give the necessary background to understand how to get from the F* specification to the actual C code, and the guarantees expected from the code. In section 9.3, we propose Dragonstar as a long-term mitigation to prove the absence of large classes of vulnerabilities by design. We describe the common cryptographic API used by hostap and how we implement it by leveraging the formally verified HACL* cryptographic library. Finally, in section 9.4, we discuss the limitations of our contribution.

Takeaway:

This contribution supports the following conclusions:

- ➤ Existing solutions enable providing more sustainable solutions, instead of simple patches.
- ➤ Formally verified implementation of the cryptographic operation of Dragonfly offers decent performance compared to OpenSSL.

As this contribution specifically targets an implementation aspect of Dragonfly, it is recommended that the reader is familiar with the protocol. To get a better grasp of our motivation, we also suggest learning about recent side-channel attacks on its implementation that we presented in previous chapters. The appropriate background to grasp the details of our contribution is provided in:

- ➤ Chapter 3, especially section 3.3, which gives an overview of Dragonfly as a protocol.
- ➤ Chapter 7 for a description of our attacks on Dragonfly implementations. Especially, in section 7.5, we discuss the limitation of current mitigation strategies.

On the other hand, the background on generating verified code and HACL* is included in this chapter, in section 9.2.

Contents

9.1. Context and Motivations 139
9.2. Formally Enforcing Secret Independence 140
9.2.1. Available Solutions 140
9.2.2. HACL*: A Verified Cryptographic Library 140
9.3. Dragonstar: Formally Verified Cryptography for Dragonfly . . 142
9.3.1. Code Structure of hostap 142
9.3.2. Using Verified Cryptography from HaCl* 143
9.3.3. Benchmark 144
9.4. Discussions 145

9.1. Context and Motivations

It is now publicly acknowledged that side channels seriously threaten cryptographic implementation. We illustrated the practicality of this attacks in chapter 6 and chapter 7, and discussed the concerns in section 8.1. In chapter 8, we established that developers involved in cryptography-related projects are aware of this threat. However, most do not leverage computer-aided verification as part of the developing process.

This is mainly due to usability issues with the tools. Consequently, many side channels still plague modern implementations for various reasons. It may be due to improper management of security operations in complex code bases: one may think of the `BN_FLG_CONSTTIME` from OpenSSL, at the root of the vulnerability in chapter 6, and previously causing side channels in DSA [GHT+20; GBY16]. Even in more recent implementations, bugs may still slip through the manual analysis of developers, as it was the case for Dragonfly implementations.

Considering that iterative analysis of Dragonfly implementation continues to reveal new leakage sources, we decided to drop the hack-and-patch approach to offer a better solution.

Limitations of automated analysis. Our first try was to leverage an automated tool to help analyze the implementation of hostap and fix all leakages we could find. Unfortunately, formal verification tools suffer from scalability issues: they are most helpful when used during the development to verify small pieces of code. However, formally verifying large implementations is daunting if we must dig deeper each time an abstraction layer is defined (*e.g.,* a call to a cryptographic library). Namely, a tool providing complete coverage, formal guarantees, and scalability is yet to be seen.

Some tools, such as timecop [Nei], perform simple taint analysis, dropping the formal guarantees for scalability. We tried running such a basic tool, that claims to be neither sound nor complete, on hostapd v2.9 with OpenSSL 1.1.1l (see chapter 7). Unfortunately, a simple Dragonfly handshake, with the password tainted as secret, raise thousands of warnings. Some represent an "obvious" secret dependence, but others may be buried in intricate computations, making it difficult to tie back to the original secret. Considering their number and intricacy, assessing their exploitability is arduous, which motivated us to provide a secure alternative.

Enforcing secret independence. Most of the warnings raised were located in the cryptographic library, which is known to be flawed [WSBS20]. This motivated us to provide a secure alternative. Instead of verifying existing implementations, we tackle the issue from the other end and build a formally verified implementation of the cryptographic operation needed to implement Dragonfly. Contrary to software verification, the idea is not to verify existing code or binary, but to *enforce* secret independence, *e.g.,* using a strong type system or domain-specific languages. This discipline is also known as high-assurance cryptography.

We do not aim to demonstrate a *potential* solution to the problem by creating an unmaintained academic Proof of Concept. Instead, we want to integrate a formally verified mitigation into the hostap code base so that it becomes a valid alternative for anyone. To this end, we must consider the practical side of our contribution, be it optimizing the implementation, providing easy-to-maintain code, or complying with the existing project structure.

To our knowledge, HACL* [ZBPB17; PPF+20; PBP+20] (and more generally the Project Everest) is the only verified cryptographic library that includes all required algorithms (including HMAC-SHA256 and NIST P-256) and verifies both functional correctness and secret independence. In addition, opting for HACL*, we benefit from its maintainers' experience to

recommend the best strategy of build/update in real projects, as they already do for Mozilla and Wireguard.

9.2. Formally Enforcing Secret Independence

> **Reminder:** *Secret independence* is the coding discipline that enforces a constant control flow and data flow on secret values. It is achieved by ensuring that (i) there is no data flow from secrets into branch conditions; (ii) addresses used for memory access do not depend on secret data; (iii) no secret-dependent data is used as input to variable-time arithmetic instructions.

Many works have advocated using formal verification to guarantee the correctness and security of cryptographic libraries and protocol implementations (see [BBB+21] for a detailed survey). Each method relies on a formal leakage model, where the attacker is typically allowed to observe all branch results and the sequence of all memory addresses accessed by the program. A verification tool then implements a sound and conservative analysis so that if it says a program is secret independent, the attacker cannot distinguish a secret from a fresh random value.

9.2.1. Available Solutions

Fiat-Crypto [EPG+19] and Cryptoline [FLS+19] can be used to verify field arithmetic (big number) functions in C and assembly for correctness, but not for secret independence. The SAW workbench [Tom16] and Coq prover [SVWW21] have been used to verify full graphic algorithms in C and Java, including selected elliptic curves, for correctness but not for secret independence. Vale [BHK+17] and Jasmin [ABB+17] have been used to verify both the correctness and secret independence of some cryptographic algorithms in Intel assembly. HACL* [ZBPB17] enforces secret independence on cryptographic implementations in C.

The approach based in assembly and C-code can be similar, to the point that they are not mutually exclusive. In fact, the Project Everest combined HACL* and Vale to build a library, called Evercrypt [PPF+20], from verified C code and verified Intel assembly.

For practical reasons, we chose the last solution to build our secure implementation for Dragonfly. More specifically, we used the API from HACL* to provide the best portability. Hereafter, we give further details on the guarantees it provides and how it achieves high-assurance cryptography.

9.2.2. HACL*: A Verified Cryptographic Library

HACL* is a cryptographic library [ZBPB17; PPF+20; PBP+20] that includes formally verified C implementations for a full suite of modern cryptographic algorithms, including hash functions, encryption algorithms, elliptic curves, and signature schemes, all of which are verified for the following properties:

- Functional correctness: the code behavior comply with mathematical functions derived from the official standards.
- Memory safety: the generated C code is free from common memory bugs such as buffer overflows, use-after-free, ...

- Secret independence: interactions between variables and operations are restricted to avoid secret dependence.

HACL* is entirely written in Low*, a subset of the F* language. The Low* implementation is then proved and compiled to C code.

From F* to C code. F* is a general-purpose functional programming language [SHK+16; Swa22] that aims at providing the programmer with rich, expressive, and flexible semantics so that they can implement and verify anything and yet rely on a substantial level of automation thanks to interactions with automated solvers. This language is particularly relevant to proving the functional correctness of cryptographic implementations, as operations are defined as pure mathematical functions.

Once the specification of a function is implemented, it can be proved functionally correct by combining Satisfiability Modulo Theories (SMT) solvers, symbolic computation, and F* tactics. Essentially, F* collects all the facts that have to be proven (*e.g.,* the absence of arithmetic overflow or division by zero) and encodes them as first-order logic operations to be fed to the SMT-solver (Z3[1] for F*). The solver will then automate the deductive verification of these facts. Programmers may adapt F* code to enable symbolic computation and simplify the solver's state (essentially, it comes down to instantly computing fixed values instead of dragging complex equations in the solver). Similarly, F* *tactics* can be used to manipulate the proof state by applying propositional logic.

Low* [PZR+17] is a subset of F* that implements a C-like memory model which enables explicit and strict memory management on the stack and the heap. Hence, Low* programs must obey restrictions, and the programmer must use explicitly heap or stack-allocated arrays and manage their lifetimes manually, and similarly rely on Low* libraries for machine integers, endianness, ... Low* code then benefits from both worlds: it can be proved as any F* program and ensure memory safety and proper heap/stack management. Then, KreMLin can be used to compile Low* to C code.

Enforcing secret independence. When implementing a cryptographic algorithm, the programmer annotates each bytestring and integer in the program as secret or public. This F* type system tracks all secret and public bytes and integers to enforce a secret-independent discipline. For example, each encryption key is treated as an array of secret (opaque) bytes. The programmer may convert these bytes into secret integers, perform secret-independent operations like additions and multiplications on secret integers, and convert them back to bytes. However, they cannot compare secret bytes (or secret integers) and cannot use secret integers as array indexes. A security theorem then states that this type-based discipline enforces secret independence [PZR+17].

A detailed example of the Low* specification for the elliptic curve point decompression function is provided in Listing 9.1. The example involves both secret and memory management, illustrating our use case.

[1] https://github.com/Z3Prover/z3

9.3. Dragonstar: Formally Verified Cryptography for Dragonfly

At the heart of the attacks we discussed in section 7.3 is the use of cryptographic libraries that are not secret-independent. To address this root cause, we built a plugin for hostap, where all the cryptographic calls within Simultaneous Authentication of Equals (SAE) redirect to the HACL* formally verified library. Below, we briefly outline our implementation and its use of HACL*.

9.3.1. Code Structure of hostap

As most projects of such size, hostap defines multiple abstraction layers, and is organized as modules, as depicted in Figure 9.1. Of particular importance, all cryptographic operations are processed through a common cryptographic API. This design allows a very modular approach, and eases the support of new libraries. Adding a new library can be done by implementing an interface between the common API and the underlying library for all required functions.

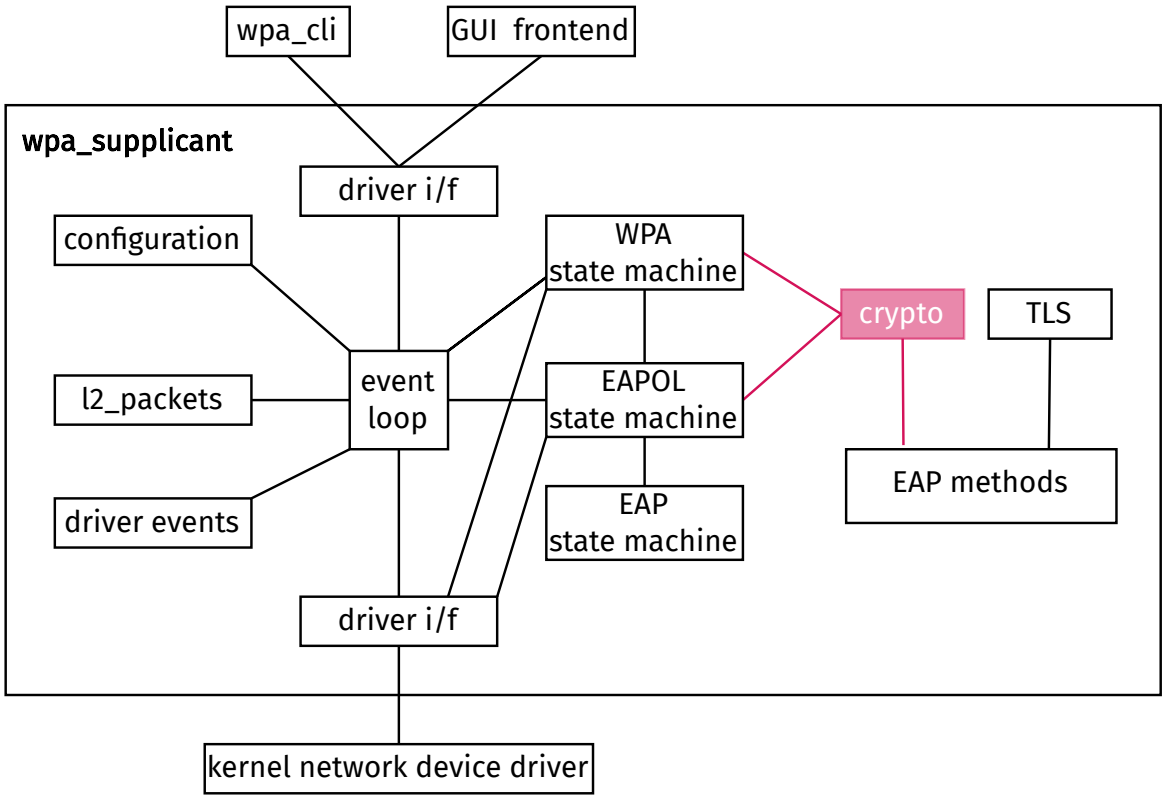


Figure 9.1. – Simplified structure of wpa_supplicant modules. The cryptographic module is represented in red.

> **Reminder:** Dragonfly must support NIST's P256 (section 3.3).

We provided an implementation of the cryptographic API using functions from the HACL* crypto library. In particular, we use verified code for the HMAC-SHA256 message authentication code, the NIST P-256 elliptic curve, and the generic Bignum library. To provide support

```
val decompress: input: lbuffer uint8 33ul → result: lbuffer uint8 64ul → Stack bool
  (requires fun h → live h input ∧ live h result ∧ disjoint input result)
  (ensures fun h0 success h1 →
    let compressed = as_seq h0 input in
    let uncompressed = as_seq h1 result in
    (uncompressed,success) == decompress_spec compressed ∧ modifies (loc result) h0 h1)
```

Listing 9.1 – F* type for the point decompression.

to the required API, we worked to expose several internal HACL* functions. In total, we wrap 28 verified functions from HACL* and the code to meet the hostap cryptographic API.

Our implementation is still being appropriately integrated into hostap project. In the meantime, a prototype integration of our work is available online[2]. We extracted the module in charge of executing the SAE handshake, and switched the cryptographic provider to call our modified branch[3] of HACL* instead of OpenSSL.

9.3.2. Using Verified Cryptography from HaCl*

All the code we use from HACL* is verified for correctness, memory safety, and secret independence. For example, and related to the vulnerability described in section 7.3, the point decompression function in HACL* is verified to have the F* type given in Listing 9.1.

This function takes a compressed point (`input`) and decompresses it into `result`, returning a boolean indicating success or failure. Both the input and the result are fixed-length arrays (`lbuffer`) that are assumed to contain secret bytes, indicated by the type `uint8`. By default, we treat all bytes and integers as secrets; if an array is known to contain only public bytes, we would use `pub_uint8` instead of `uint8`. Hence, the type given to `input` constrains the code of the `decompress` function to treat the contents of `input` and any value derived from `input` as opaque secret values. Branching on the parity of the y-coordinate, for example, would result in a type error, since secret bytes do not have a comparison operation. In other words, the type of `input` ensures that the code must be secret-independent with respect to its contents.

In addition to secret independence, the type above also enforces memory safety and correctness. The function is in the `Stack` effect, indicating that it only uses the stack and does not allocate or free any memory in the heap. The precondition (`requires`) states that the `input` and `result` arrays point to valid disjoint locations in the heap. The post-condition (`ensures`) says that the output of `decompress` matches its spec `decompress_spec` and that the function only modifies the `result` array.

Similarly to point decompression, the big number conversion function in HACL* (Listing 9.2) is verified to be secret independent. In particular, it does not strip leading zeroes and produces a padded big number given a specific maximum size.

Consequently, by using verified functions from HACL*, we eliminate the two leaks we have explored in this paper, and more generally, we formally guarantee the absence of a large class of timing attacks on our code. Related limitations are discussed in section 9.4.

[2] https://gitlab.inria.fr/ddealmei/artifact_dragonstar
[3] https://github.com/project-everest/hacl-star/tree/dragonstar_temp

```
val bn_from_bytes_be: #t:limb_t
  → len:size_t{0 < v len ∧ numbytes t * v (blocks len (size (numbytes t))) ≤ max_size_t}
  → input:lbuffer uint8 len
  → result:lbignum t (blocks len (size (numbytes t))) →
  Stack unit
  (requires fun h → live h input ∧ live h result ∧ disjoint input result)
  (ensures fun h0 _h1 →
    as_seq h1 res == Spec.bn_from_bytes_be (v len) (as_seq h0 b) ∧
    modifies (loc result) h0 h1)
```

Listing 9.2 – F* type for the big number conversion.

9.3.3. Benchmark

All benchmarks have been performed using the tool `perf` on the same set of inputs, while fixing all used random values. All benchmarks were performed on a Dell XPS 13 7390 running on Ubuntu 20.04.2, kernel 5.13.0-39, with an Intel(R) Core(TM) i7-10510U and 16 GB of RAM.

> **Reminder:** An SAE session is established based on a secret password and the Media Access Control (MAC) addresses of the two peers. For SAE-PT, an additional identifier is involved (section 3.3).

We evaluated the performance by extracting the SAE module from hostap (`common/sae.c`) and testing different cryptographic providers. We measured the handshake performance for multiple passwords for each provider while fixing the other parameters.

We repeated experiments on 20 different passwords, with fixed MAC addresses and password identifier. Then, we computed the number of cycles required to establish 1,000 sessions with each password. For SAE-PT, we generated the point PT once for each password, and reused the pre-computed value in all subsequent session establishment, as intended by the standard. Thus, the initial cost of computing PT is smoothed by the repeated sessions.

Figure 9.2 represents the average number of cycles needed to perform a Dragonfly handshake using hostap for both SAE and SAE-PT. Here, we compare our HACL*-based implementation with OpenSSL, which is the default cryptographic library in most settings. We also include the OpenSSL build with no assembly code (*i.e.,* `noasm`).

We highlight two main findings. First, SAE-PT is always faster than SAE. This can be explained by the fact that PT is only computed once. In addition, the mitigations implemented by hunting-and-pecking require to loop over the conversion a fixed number of times, implying dummy unneeded iterations. In contrast, Simplified Shallue-van de Woestijne-Ulas (SSWU) offers a linear workflow with a single conversion, and arithmetic optimizations. Second, HACL* constitutes a good alternative to OpenSSL. Indeed, HACL* is not only formally proved to be secret-independent, but also provides decent efficiency. It outperforms OpenSSL `noasm` for both SAE and SAE-PT. However, it is slower when assembly code leveraging specialized CPU instructions is used.

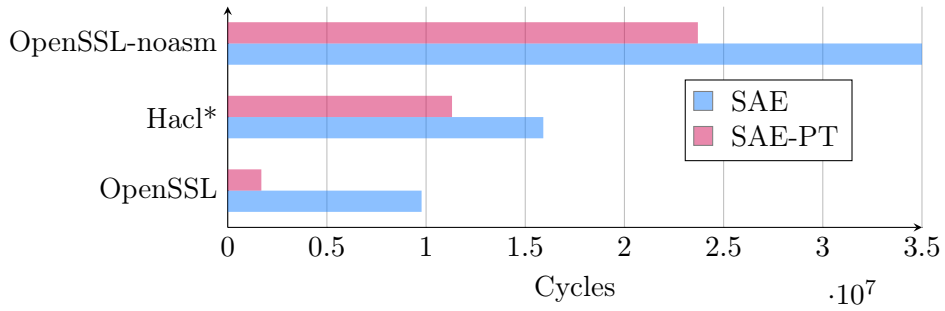


Figure 9.2. – Performance comparison (in Central Processing Unit (CPU) cycles) of our implementation and OpenSSL implementation. All results are obtained by repeating the Dragonfly handshake 1000 times for 20 different passwords. For SAE-PT, we compute PT only once per password, as intended by the specification.

9.4. Discussions

Leftover Side Channels. As many guarantees as HACL* brings, secret independence does not eliminate all side-channel attacks. In particular, HACL* and our code may still be vulnerable to fault-injection or transient execution attacks. New approaches for verifying code against advanced side-channels are still under active development [BCG+21; BBC+19]. For example, the domain-specific language FaCT [CSJ+19b] recently evolved to provide guarantees against transient execution attacks [SBB+22]. If and when mitigations and verification techniques for such attacks are incorporated into HACL*, our implementation will automatically benefit from these defenses. We also note that we do not yet provide a full proof of correctness for Dragonstar with respect to the Dragonfly specification. Instead, only the cryptographic provider is proved secure, so other flaws can still sneak into the base code. However, our approach does come with a significant practical advantage since it allows smooth integration into existing Wi-Fi daemons. Indeed, Dragonstar fits the cryptographic-provider structure of hostap, requiring only a minimal change to the existing project. Because the password conversion methods are implemented in the core of hostap, a verified implementation of such routines would have resulted in consequent changes, and integration of generated code which is hard to understand and maintain.

Compiler-induced Side Channels. The observable properties of a program's execution are generally not evident from source code alone. Thus, software-invisible optimizations (*e.g.,* compiler optimizations or data-dependent Instruction Set Architecture (ISA) optimizations) can remove source-level countermeasures. To avoid such an event, we advise using a trusted compiler, such as CompCert [Ler09], that preserves the semantics of the source program. However, this additional verification pass may hinder the performance of the resulting binary and result in a binary less efficient than one compiled with clang or gcc.

Part III.

Conclusion and Future Work

Conclusion

Side-channels have become increasingly practical in the last decades. However, real-world applications are only starting to consider them as a significant threat and prefer to concentrate their engineering efforts elsewhere. In this thesis, we help to bridge the gap between academic and real-world solutions. Our contributions are twofold, working both on the offensive and defensive sides.

Attacks on PAKEs. We started by investigating attack vectors related to microarchitectural side channels, unveiling new leakage in widely deployed implementations of Password-Authenticated Key Exchange protocols (PAKE)s. The choice of this protocol class is motivated by their renewed interest and recent integration in new standards, foreshadowing a massive deployment [All21]. Then, for each vulnerability, we demonstrated its practical impact by implementing a full practical Proof of Concept, hoping to make projects consider these threats more seriously.

Ultimately, we discovered multiple vulnerabilities in both Secure Remote Password protocol (SRP) and Dragonfly, each leading to a full password recovery attack because the control flow of their execution is secret-dependent. The study of these implementations boils down to the following takeaways:

- OpenSSL implementation is still riddled with side-channel, particularly because of their flag-based protection. This issue has been extensively studied and criticized by Pereida García [Per22]. Our work on SRP shows once again that their insecure-by-default approach is prone to misuses and errors.

- Newer implementations are no better. WPA3 suffers from poor decision-making to begin with (despite strong opposition from CFRG members), which has led to numerous implementation pitfalls. Hence, first Dragonfly implementations were prone to side channel, and required hacks to get around the probabilistic nature of the password conversion method. It took a couple of years to the Wi-Fi Alliance to standardize a (non-backward compatible) alternative. Now, Dragonfly still supports the legacy method. Moreover, despite the constant-time design of the new solution, we showed that peculiar interaction with cryptographic libraries led to unexpected side-channel leakages.

The vulnerabilities we found are easily fixed by switching to a secret-independent control flow (*e.g.,* setting a flag for SRP). However, we faced the limitations of current mitigations strategies: considering the arduous task that is verifying an implementation against side channels, simple patches are not merely enough to secure implementations. A better alternative

would be a systematic analysis of the implementation, with a formal guarantee that it is secret-independent.

Toward Better Mitigations. Our attempts to formally verify large implementations using existing tools were unsuccessful due to scalability issues. However, including the verification within the development process would make it more suitable and easier to scale. Hence, we surveyed real-world project developers to understand the lack of computer-aided verification in the development process. From their responses, we deduced that while they are aware of side channels, most of them find that using formal tools requires too much effort. Hence, they use uncomplete/unsound tools, if any. In addition, academic tools often lack support and documentation, and may not be suited to real-world constraints, such as usage in a continuous integration pipeline. As a result, we defined a list of recommendations for tool developers, project maintainers, and compilers developers that should improve the deployment of computer-aided verification and, hopefully, reduce the spread of side-channel leaks.

Since we were aiming for a sustainable solution to the side-channel leaks in Dragonfly implementations, we explored another lead: high-assurance cryptography. Instead of verifying existing implementation and investigating every potential constant-time violation, we generated a formally verified implementation using HACL*. We did not aim at a simple example, but to deploy it in the hostap project as an alternative cryptographic provider. We ended up with a formally verified implementation of the cryptographic layer of Dragonfly, that is being integrated into hostap.

Answering our research questions. At the beginning of this thesis (section 1.1), we defined research questions that our contributions addressed. Here is a succinct answer to each of them:

I. *Do PAKE protocols include specific protections against side-channels, considering their particular assumptions?*
It is important to distinguish protocols from their implementations. For the former, despite its recency, the early standard for Dragonfly reflected a lack of rigor regarding side channel prevention. However, despite these questionable choices, the protocol quickly evolved to recommend a more secure primitive with a secret-independent workflow. We note that For the latter, considering the attacks we present in chapter 6 and chapter 7, we conclude that widespread implementations are not secure yet, despite an improvement on the standard choices. On the other hand, many developers are responsive and eager to make their implementation more secure. Along with academic effort, we are shifting to better overall security. However, there is hope in the future, as recently standardized PAKEs are less *prone* to side channels. Nevertheless, their implementations still need to be verified.

II. *Why is today's cryptographic software not free of timing-attack vulnerabilities?*
We identify a discrepancy between the academics providing many computer-aided verification solutions, and real-world developers not using them. It seems that the main issue arises from a usability aspect: academics mostly provide tools to illustrate an approach, but the tool benefits from low to no maintenance and cannot be easily integrated into the development process. Meeting the developers where they are, and

providing easy-to-use tools that can be integrated into continuous integration, would speed up their deployment.

III. *How can we provide guarantees against side-channel attacks at the software level?* The previous question reflects the first solution: integrating a systematic verification in the development process would significantly reduce the number of side channels. Another approach that we took for Dragonfly (chapter 9), is to rely on high-assurance cryptography. We showed that solutions already exist and can be adapted to the particular need of a protocol to ensure that there is no side channel left.

Perspectives and Current Work

This section provides an insight into ongoing or future works, expanding on previous results and following different leads.

Secure Next Generation of PAKEs

The Crypto Forum Research Group (CFRG) standardization process ended in 2021, with two new recommended Password-Authenticated Key Exchange protocols (PAKE): OPAQUE [JKX18] and CPace [HL19]. The new standards are still in draft [AHH22; BKLW22], but implementations are already appearing in Go[4], Rust[5] and C[6, 7] and are likely to spread soon. As a requirement of the competition, both protocols benefit from security proof. However, as we demonstrated in this thesis, implementations can still introduce leakage vectors and must be adequately analyzed.

Since it is still early in their standardization and deployment, we have the opportunity to *prevent* side-channels from being deployed in production systems, instead of fixing them. This can lead to two interesting contributions: (i) a systematic analysis of the early implementations of the protocols; (ii) the creation of an optimized and formally verified implementation to be used as a reference for future deployment.

Enhancement of Flush+Reload's Spatial Resolution

Flush+Reload is excellent for instruction-driven cache attacks, but its theoretical spatial resolution of one cache line suffers from Central Processing Unit (CPU) optimizations such as prefetching instruction, hindering its performance. To avoid stalling while waiting for instructions to be fetched in the cache, and processed by the execution pipeline, modern CPUs resort to various optimization such as out-of-order or speculative execution and prefetching. This means that a particular cache line may be fetched in the cache by these execution units *before* the process reaches this point of the execution. This may lead to false cache hits, degrading the precision of the attacker's measurements.

Instead of suffering from these optimizations, we took advantage of them and redesigned the workflow of Flush+Reload. Instead of probing the spied instructions, we probed one cache line after while heavily degrading the performance of the instructions whose execution we wanted to detect. The goal is to create a race between the Performance Degradation Attack

[4] https://github.com/bytemare/opaque/
[5] https://github.com/novifinancial/opaque-ke
[6] https://github.com/stef/libopaque
[7] https://github.com/aldenml/ecc

(PDA) and the prefetcher. The PDA will make the target instruction long enough to execute so that the temporal resolution of Flush+Reload is reached, and the attacker detects multiple cache-hit caused by the prefetcher. This technique provides a better spatial resolution than the classical Flush+Reload, allowing the attacker to spy on less than a single cache line of instructions. This improvement enables the attacker to exploit Side-Channel Attack (SCA) that were previously left aside because considered "*not practical*". We were able to implement Proof of Concept (PoC) attacks on multiple sensitive functions of OpenSSL to demonstrate our results. Furthermore, we exploited it in a practical scenario to leak information during the Dragonfly handshake (section 7.3). However, the exact mechanisms responsible for this leakage are still to be determined, and additional testing is needed to understand the impact better.

Tool-Assisted SCA Exploitation

Besides the recent increase in microarchitectural attack discovery, the complexity of their implementation makes a steep learning curve and can be a barrier to entering this field. Even for experienced researchers, the investigation phase can be tedious and significantly slows down research. For instance, exploring the binary and testing different probing locations for Flush+Reload (or related) attacks may be time-consuming and repetitive. Nonetheless, it remains an essential part of the practical exploitation of such attacks. Approaches such as Cache Template Attack [GSM15] can be beneficial but reach some limitations when the exploitation of a vulnerability implies a combination of multiple attacks and stray further from the naive use case.

It would be interesting and beneficial to provide an easy way to investigate binaries and locate suitable locations to spy on in the preliminary investigation phase. We started to work on a GUI tool automatically using multiple existing CLI utilities to make binary exploration easier, allowing us to quickly locate memory addresses of targeted instruction, as long as their distance (in byte or cache lines). This work is still in progress, and while the first results are encouraging, it is not publicly available yet.

Usability Assessment of Various Constant-Time Verification Tools

> **Reference:** This is based on a joint work with Jàn Jancar, Marcel Fourné, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. It expends our previous work [JFB+22] but is still on-going.

In our previous work [JFB+22], we identify various aspects of existing tools preventing developers from using them in practice. We believe that identifying the usability issues may help current and future tool developers to better design their tools.

In this work, we aim to find the culprit making non-expert users reluctant to use constant-time verification tools. We are conducting a survey graduate/Ph.D. students with moderate experience in coding and meager experience in SCA analysis. We asked them to use different tools on both toy examples and real-world cryptographic libraries. After each task, they complete a survey to give us feedback on what made solving the task easy/difficult. The survey is still in process, so we cannot conclude yet.

Bibliography

[AB22] Alejandro Cabrera Aldaya and Billy Bob Brumley. *HyperDegrade: From GHz to MHz Effective CPU Frequencies*. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. Link. (Cit. on p. 27).

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. *Jasmin: High-Assurance and High-Speed Cryptography*. In: *CCS*. ACM, 2017, pp. 1807–1823 (cit. on pp. 116, 140).

[ABB+20a] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. *The Last Mile: High-Assurance and High-Speed Cryptographic Implementations*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 965–982 (cit. on p. 116).

[ABB+20b] Melissa Azouaoui, Davide Bellizia, Ileana Buhan, Nicolas Debande, Sébastien Duval, Christophe Giraud, Éliane Jaulmes, François Koeune, Elisabeth Oswald, François-Xavier Standaert, and Carolyn Whitnall. *A Systematic Appraisal of Side Channel Evaluation Strategies*. In: *Security Standardisation Research - 6th International Conference, SSR 2020, London, UK, November 30 - December 1, 2020, Proceedings*. Ed. by Thyla van der Merwe, Chris J. Mitchell, and Maryam Mehrnezhad. Vol. 12529. Lecture Notes in Computer Science. Springer, 2020, pp. 46–66. Link. (Cit. on p. 109).

[ABF+16] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. *Amplifying side channels through performance degradation*. In: *ACSAC*. ACM, 2016, pp. 422–435 (cit. on p. 27).

[ABF+17] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. *Comparing the Usability of Cryptographic APIs*. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 154–171 (cit. on p. 116).

[ABH+19] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. *Port Contention for Fun and Profit*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 870–887 (cit. on p. 23).

[ABPV13] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. *"Formal verification of side-channel countermeasures using self-composition"*. In: *Sci. Comput. Program.* 78.7 (2013), pp. 796–812. Link. (Cit. on pp. 112, 113, 126).

[ACE+18] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. *SideTrail: Verifying Time-Balancing of Cryptosystems*. In: *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*. Ed. by Ruzica Piskac and Philipp Rümmer. Vol. 11294. LNCS. Springer, 2018, pp. 215–228. Link. (Cit. on pp. 112, 113, 126).

[AF18] Gildas Avoine and Loïc Ferreira. *"Attacking GlobalPlatform SCP02-compliant Smart Cards Using a Padding Oracle Attack"*. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (May 2018), pp. 149–170. Link. (Cit. on pp. xvii, 10).

[AG16] Proton Technologies A.G. *ProtonMail Security Features and Infrastructure*. 2016. Link. (Cit. on p. 72).

[AGH+17] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. *Decomposition instead of self-composition for proving the absence of timing channels*. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 362–375. Link. (Cit. on pp. 112, 113, 126).

[AGTB19] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. *"Cache-Timing Attacks on RSA Key Generation"*. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.4 (2019), pp. 213–242 (cit. on p. 27).

[AHH22] Michel Abdalla, Björn Haase, and Julia Hesse. *CPace, a balanced composable PAKE*. Internet-Draft draft-irtf-cfrg-cpace-06. Work in Progress. Internet Engineering Task Force, July 2022. 74 pp. Link. (Cit. on p. 153).

[All18] Wi-Fi Alliance. *WPA3 Specification Version 1.0*. Tech. rep. Accessed: 2022-07-22. Wi-Fi Alliance, 2018. Link. (Cit. on pp. 34, 37).

[All19] Wi-Fi Alliance. *WPA3 Security Considerations*. Nov. 2019 (cit. on pp. 79, 81).

[All20a] Wi-Fi Alliance. *Wi-Fi CERTIFIED Certification Overview*. Mar. 2020 (cit. on pp. 34, 79).

[All20b] Wi-Fi Alliance. *WPA3 Specification Version 3.0*. Tech. rep. Accessed: 2022-07-22. Wi-Fi Alliance, 2020. Link. (Cit. on p. 37).

[All21] Wi-Fi Alliance. *Global economic value of Wi-Fi*. Accessed: 2022-07-22. Sept. 2021. Link. (Cit. on pp. xi, 5, 79, 149).

[Ama20] Amazon. *Amazon EC2 P4d Instances*. 2020. Link. (Cit. on p. 67).

[AMPS22] M. R. Albrecht, L. Mareková, K. G. Paterson, and I. Stepanovs. *Four Attacks and a Proof for Telegram*. In: *2022 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 223–242. Link. (Cit. on p. 46).

[ANT+20] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. *LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage*. In: *CCS*. ACM, 2020, pp. 225–242 (cit. on p. 27).

[AP05] Michel Abdalla and David Pointcheval. *Simple Password-Based Encrypted Key Exchange Protocols*. In: *CT-RSA*. Vol. 3376. Lecture Notes in Computer Science. Springer, 2005, pp. 191–208 (cit. on p. 31).

[AP16] Martin R. Albrecht and Kenneth G. Paterson. *Lucky Microseconds: A Timing Attack on Amazon's* s2n *Implementation of TLS*. In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. LNCS. Springer, 2016, pp. 622–643. Link. (Cit. on p. 111).

[App20] Apple. *HomeKit Accessory Development Kit (ADK)*. Accessed: 2022-07-22. 2020. Link. (Cit. on pp. 32, 53, 71).

[App21] Apple. *Escrow security for iCloud Keychain*. Accessed: 2022-07-22. 2021. Link. (Cit. on pp. 31, 53).

[Ass18] Hamza Assyad. *Now generally available: Amazon CognitoAuthentication Extension Library*. Accessed: 2022-07-22. 2018. Link. (Cit. on pp. 31, 53).

[BB03] David Brumley and Dan Boneh. *Remote Timing Attacks are Practical*. In: *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. ACM, 2003. Link. (Cit. on pp. x, 4, 111).

[BBB+21] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. *SoK: Computer-Aided Cryptography*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 777–795 (cit. on pp. xii, xv, 5, 8, 105, 109, 110, 112, 114, 135, 140).

[BBC+14] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. *System-level Non-interference for Constant-time Cryptography*. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM, 2014, pp. 1267–1279. Link. (Cit. on pp. 112–114, 126).

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. *maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults*. In: *ESORICS (1)*. Vol. 11735. Lecture Notes in Computer Science. Springer, 2019, pp. 300–318 (cit. on p. 145).

[BBE+19] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Mélissa Rossi, and Mehdi Tibouchi. *GALACTICS: Gaussian Sampling for Lattice-Based Constant- Time Implementation of Cryptographic Signatures, Revisited*. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*. ACM, 2019, pp. 2147–2164. Link. (Cit. on p. 112).

[BBG+17] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. *Sliding Right into Disaster: Left-to-Right Sliding Windows Leak*. In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 555–576 (cit. on p. 27).

[BBM00] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. *Public-Key Encryption in a Multi-user Setting: Security Proofs and Improvements*. In: *EUROCRYPT*. Vol. 1807. Lecture Notes in Computer Science. Springer, 2000, pp. 259–274 (cit. on p. 42).

[BC06] Virginia Braun and Victoria Clarke. *"Using thematic analysis in psychology"*. In: *Qualitative research in psychology* 3.2 (2006), pp. 77–101 (cit. on p. 119).

[BCG+21] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. *High-Assurance Cryptography in the Spectre Era*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 1884–1901 (cit. on p. 145).

[BCI+10] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. *Efficient Indifferentiable Hashing into Ordinary Elliptic Curves*. In: *CRYPTO*. Vol. 6223. Lecture Notes in Computer Science. Springer, 2010, pp. 237–254 (cit. on p. 90).

[BCP03] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. *Security proofs for an efficient password-based key exchange*. In: *CCS*. ACM, 2003, pp. 241–250 (cit. on p. 31).

[BCR+19] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, Richard Davis, and Scott Simon. *Recommendation for Pair-Wise Key Establishment Using Integer Factorization Cryptography*. Tech. rep. 800-56B Rev. 2. NIST, Mar. 2019 (cit. on pp. xvii, 10).

[BD15] Elaine B. Barker and Quynh Dang. *Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance*. Tech. rep. SP800-57 Part 3 Rev.1. NIST, Jan. 2015 (cit. on p. 58).

[BDG12] Musard Balliu, Mads Dam, and Gurvan Le Guernic. *ENCoVer: Symbolic Exploration for Information Flow Security*. In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 30–44. Link. (Cit. on p. 114).

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. *On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)*. In: *EUROCRYPT*. Vol. 1233. Lecture Notes in Computer Science. Springer, 1997, pp. 37–51 (cit. on p. 46).

[BDQG21] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. *"Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization"*. In: *CoRR* abs/2104.10749 (2021). Link. (Cit. on p. 114).

[Ber05] Daniel J. Bernstein. *Cache-timing attacks on AES*. 2005. Link. (Cit. on p. 111).

[Ber07] Antonia Bertolino. *Software testing research: Achievements, challenges, dreams.* In: *Future of Software Engineering (FOSE'07).* IEEE. 2007, pp. 85–103 (cit. on p. 125).

[BF18] Richard Barnes and Owen Friel. *Usage of PAKE with TLS 1.3.* Internet-Draft draft-barnes-tls-pake-04. Work in Progress. Internet Engineering Task Force, July 2018. 11 pp. Link. (Cit. on p. 69).

[BFK+12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. *Efficient Padding Oracle Attacks on Cryptographic Hardware.* In: *CRYPTO.* Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 608–625 (cit. on pp. xvii, 10).

[BFS20a] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. *Dragonblood is Still Leaking: Practical Cache-based Side-Channel in the Wild.* In: *ACSAC.* ACM, 2020, pp. 291–303 (cit. on pp. xiv, xvi, xxi, 7–9, 27, 90).

[BFS20b] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. *"The Long and Winding Path to Secure Implementation of GlobalPlatform SCP10".* In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 196–218 (cit. on pp. xvii, xxi, 10).

[BFS21] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. *PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild.* In: *CCS.* ACM, 2021, pp. 2497–2512 (cit. on pp. xiii, xxi, 7, 27).

[BGA+16] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. *Computer-Aided Verification for Mechanism Design.* In: *Web and Internet Economics - 12th International Conference, WINE 2016, Montreal, Canada, December 11-14, 2016, Proceedings.* Ed. by Yang Cai and Adrian Vetta. Vol. 10123. Lecture Notes in Computer Science. Springer, 2016, pp. 279–293. Link. (Cit. on pp. 112, 113, 126).

[BH95] Yoav Benjamini and Yosef Hochberg. *"Controlling the false discovery rate: a practical and powerful approach to multiple testing".* In: *Journal of the Royal statistical society: series B (Methodological)* 57.1 (1995), pp. 289–300 (cit. on p. 119).

[BHK+17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. *Vale: Verifying High-Performance Cryptographic Assembly Code.* In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 917–934. Link. (Cit. on p. 140).

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. *Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme.* In: *CHES.* Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 323–345 (cit. on p. 112).

[BHP22] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. *MEGA: Malleable Encryption Goes Awry.* To appear in: 44th IEEE Symposium on Security and Privacy, 2023. June 2022. Link. (Cit. on p. 46).

[BJK15] Mihir Bellare, Joseph Jaeger, and Daniel Kane. *Mass-surveillance without the State: Strongly Undetectable Algorithm-Substitution Attacks*. In: *ACM Conference on Computer and Communications Security*. ACM, 2015, pp. 1431–1440 (cit. on pp. xvii, 9).

[BKLW22] Daniel Bourdrez, Dr. Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. *The OPAQUE Asymmetric PAKE Protocol*. Internet-Draft draft-irtf-cfrg-opaque-09. Work in Progress. Internet Engineering Task Force, July 2022. 70 pp. Link. (Cit. on p. 153).

[BL09] Daniel J. Bernstein and Tanja Lange. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. Accessed: 2022-07-22. 2009. Link. (Cit. on p. 133).

[Ble98] Daniel Bleichenbacher. *Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1*. In: *CRYPTO*. Vol. 1462. Lecture Notes in Computer Science. Springer, 1998, pp. 1–12 (cit. on pp. xvii, 10).

[BM92] Steven M. Bellovin and Michael Merritt. *Encrypted key exchange: password-based protocols secure against dictionary attacks*. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1992, pp. 72–84 (cit. on pp. xii, 6, 30, 31).

[BM93] Steven M. Bellovin and Michael Merritt. *Augmented Encrypted Key Exchange: A Password-Based Protocol Secure against Dictionary Attacks and Password File Compromise*. In: *CCS*. ACM, 1993, pp. 244–250 (cit. on pp. 30, 31).

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. *Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman*. In: *EUROCRYPT*. Vol. 1807. Lecture Notes in Computer Science. Springer, 2000, pp. 156–171 (cit. on p. 30).

[BMS20] Colin Boyd, Anish Mathuria, and Douglas Stebila. *Protocols for Authentication and Key Establishment, Second Edition*. Information Security and Cryptography. Springer, 2020 (cit. on p. 42).

[BPH22] Lara Bruseghini, Kenneth G. Paterson, and Daniel Huigens. *Victory by KO: Attacking OpenPGP Using Key Overwriting*. Proceedings of ACM Conference on Computer and Communications Security, Los Angeles, November 2022. 2022. Link. (Cit. on p. 46).

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. *Authenticated Key Exchange Secure against Dictionary Attacks*. In: *EUROCRYPT*. Vol. 1807. Lecture Notes in Computer Science. Springer, 2000, pp. 139–155 (cit. on pp. 30, 31, 34).

[BPT17] Sandrine Blazy, David Pichardie, and Alix Trieu. *Verifying Constant-Time Implementations by Abstract Interpretation*. In: *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*. Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Vol. 10492. LNCS. Springer, 2017, pp. 260–277. Link. (Cit. on pp. 112, 113, 126).

[Bri11] Ernie Brickell. *Technologies to Improve Platform Security*. Invited talk at CHES 2011. 2011. Link. (Cit. on p. 111).

[BS97] Eli Biham and Adi Shamir. *Differential Fault Analysis of Secret Key Cryptosystems*. In: *CRYPTO*. Vol. 1294. Lecture Notes in Computer Science. Springer, 1997, pp. 513–525 (cit. on p. 46).

[BSBP18] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. *Symbolic path cost analysis for side-channel detection*. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 27–37. Link. (Cit. on pp. 112, 113, 126).

[BSI06] BSI. *Technical Guideline TR-03110 - Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token – Part 2: Protocols for electronic IDentification, Authentication and trust Services (eIDAS)*. Tech. rep. Version 1.0. Federal Office for Information and Security, 2006 (cit. on p. 31).

[BSI09] BSI. *Technical Guideline TR-03110 - Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token – Part 2: Protocols for electronic IDentification, Authentication and trust Services (eIDAS)*. Tech. rep. Version 2.0. Federal Office for Information and Security, 2009 (cit. on p. 31).

[BSY18] Hanno Böck, Juraj Somorovsky, and Craig Young. *Return Of Bleichenbacher's Oracle Threat (ROBOT)*. In: *USENIX Security Symposium*. USENIX Association, 2018, pp. 817–849 (cit. on p. 46).

[BT11] Billy Bob Brumley and Nicola Tuveri. *Remote Timing Attacks are Still Practical*. In: *Computer Security—ESORICS 2011*. Ed. by Vijay Atluri and Claudia Diaz. Vol. 6879. LNCS. Springer, 2011, pp. 355–371 (cit. on pp. 111, 131).

[Bur18] Elie Bursztein. *The bleak picture of two-factor authentication adoption in the wild*. 2018. Link. (Cit. on p. 29).

[But16] Bart Butler. *Improved Authentication for Email Encryption and Security*. Accessed: 2022-07-22. 2016. Link. (Cit. on pp. 32, 53).

[CBS+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. *A Systematic Evaluation of Transient Execution Attacks and Defenses*. In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 249–266. Link. (Cit. on p. 114).

[CDG+20] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. *Constant-time foundations for the new spectre era*. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 913–926 (cit. on p. 114).

[CDM+22] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. *SoK: Practical Foundations for Software Spectre Defenses*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2022, pp. 666–680 (cit. on p. 114).

[CFD17] Jia Chen, Yu Feng, and Isil Dillig. *Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic*. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM, 2017, pp. 875–890. Link. (Cit. on pp. 112, 113, 126).

[CFR20] CFRG. *PAKE Selection*. 2020. Link. (Cit. on pp. 30, 31, 69).

[CH14] Dylan Clarke and Feng Hao. *"Cryptanalysis of the dragonfly key exchange protocol"*. In: *IET Inf. Secur.* 8.6 (2014), pp. 283–289 (cit. on p. 42).

[CHVV03] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. *Password Interception in a SSL/TLS Channel*. In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference*. Ed. by Dan Boneh. Vol. 2729. NCS. Springer, 2003, pp. 583–599. Link. (Cit. on p. 111).

[CKP+20] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. *Pseudorandom Black Swans: Cache Attacks on CTR_DRBG*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1241–1258 (cit. on p. 27).

[CKS08] David Cash, Eike Kiltz, and Victor Shoup. *The Twin Diffie-Hellman Problem and Applications*. In: *EUROCRYPT*. Vol. 4965. Lecture Notes in Computer Science. Springer, 2008, pp. 127–145 (cit. on p. 31).

[Con98] William Jay Conover. *Practical nonparametric statistics*. Vol. 350. John Wiley & Sons, 1998 (cit. on p. 119).

[Cop96] Don Coppersmith. *Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known*. In: *EUROCRYPT*. Vol. 1070. Lecture Notes in Computer Science. Springer, 1996, pp. 178–189 (cit. on pp. xvii, 10).

[Cop97] Don Coppersmith. *"Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities"*. In: *J. Cryptol.* 10.4 (1997), pp. 233–260 (cit. on pp. xvii, 10).

[CR18] Sudipta Chattopadhyay and Abhik Roychoudhury. *"Symbolic Verification of Cache Side-Channel Freedom"*. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37.11 (2018), pp. 2812–2823. Link. (Cit. on p. 114).

[CSJ+19a] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. *FaCT: a DSL for timing-sensitive computation*. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 174–189. Link. (Cit. on pp. 112, 113, 116, 126).

[CSJ+19b] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. *FaCT: a DSL for timing-sensitive computation*. In: *PLDI*. ACM, 2019, pp. 174–189 (cit. on p. 145).

[Cuo] Pascal Cuoq. *tis-ct*. Link. (Cit. on pp. 112, 113, 126).

[DBR20] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. *Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level*. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1021–1038. Link. (Cit. on pp. 112, 113, 126).

[DFK+13] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. *CacheAudit: A Tool for the Static Analysis of Cache Side Channels*. In: *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. Ed. by Samuel T. King. USENIX Association, 2013, pp. 431–446. Link. (Cit. on pp. 112, 113, 126).

[DM21] Guillaume Didier and Clémentine Maurice. *Calibration Done Right: Noiseless Flush+Flush Attacks*. In: *DIMVA*. Vol. 12756. Lecture Notes in Computer Science. Springer, 2021, pp. 278–298 (cit. on p. 28).

[EKSS09] John Engler, Chris Karlof, Elaine Shi, and Dawn Song. *Is it too late for PAKE?* In: *W2SP*. The Internet Society, 2009 (cit. on p. 30).

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. *Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises*. In: *IEEE Symposium on Security and Privacy*. 2019, pp. 1202–1219 (cit. on p. 140).

[Fal] Rob Faludi. *Who uses Erlang for product development?* Accessed: 2022-07-22. Link. (Cit. on p. 73).

[Fal17] Rob Faludi. *Introducing the Official Digi XBee Python Library*. Accessed: 2022-07-22. 2017. Link. (Cit. on p. 74).

[Fil18] Rick Fillion. *Developers: How we use SRP, and you can too*. Accessed: 2022-07-22. 2018. Link. (Cit. on pp. 31, 53).

[FLS+19] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. *Signed Cryptographic Program Verification with Typed CryptoLine*. In: *CCS*. 2019, pp. 1591–1606 (cit. on p. 140).

[Flu14] Scott Fluhrer. *Re: [CFRG] Requesting removal of CFRG co-chair*. Jan. 2014. Link. (Cit. on pp. 34, 79).

[FP13] Nadhem J. Al Fardan and Kenneth G. Paterson. *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. In: *2013 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013, pp. 526–540. Link. (Cit. on p. 111).

[FPJ92] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. *Stride directed prefetching in scalar processors*. In: *MICRO*. ACM / IEEE Computer Society, 1992, pp. 102–110 (cit. on p. 19).

[FPS21] Luca De Feo, Bertram Poettering, and Alessandro Sorniotti. *On the (In)Security of ElGamal in OpenPGP*. In: *CCS*. ACM, 2021, pp. 2066–2080 (cit. on p. 47).

[FSS+22] Armando Faz-Hernández, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. *Hashing to Elliptic Curves*. Internet-Draft draft-irtf-cfrg-hash-to-curve-16. Work in Progress. Internet Engineering Task Force, June 2022. 175 pp. Link. (Cit. on pp. 34, 90, 105).

[GB17] Cesar Pereida García and Billy Bob Brumley. *Constant-Time Callees with Variable-Time Callers*. In: *USENIX Security Symposium*. USENIX Association, 2017, pp. 83–98 (cit. on pp. 27, 46).

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. *Cache Games - Bringing Access-Based Cache Attacks on AES to Practice*. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2011, pp. 490–505 (cit. on p. 25).

[GBY16] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. *"Make Sure DSA Signing Exponentiations Really are Constant-Time"*. In: *CCS*. ACM, 2016, pp. 1639–1650 (cit. on pp. 27, 46, 139).

[GHT+20] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. *Certified Side Channels*. In: *USENIX Security Symposium*. USENIX Association, 2020, pp. 2021–2038 (cit. on pp. 27, 46, 139).

[GJK21] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. *KHAPE: Asymmetric PAKE from Key-Hiding Key Exchange*. In: *CRYPTO (4)*. Vol. 12828. Lecture Notes in Computer Science. Springer, 2021, pp. 701–730 (cit. on p. 31).

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. *A Key-Recovery Timing Attack on Post-quantum Primitives Using the Fujisaki-Okamoto Transformation and Its Application on FrodoKEM*. In: *Advances in Cryptology – CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. LNCS. Springer, 2020, pp. 359–386. Link. (Cit. on p. 112).

[GKM+20] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. *Spectector: Principled Detection of Speculative Information Flows*. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1–19. Link. (Cit. on p. 114).

[GMR06] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. *A Method for Making Password-Based Key Exchange Resilient to Server Compromise*. In: *CRYPTO*. Vol. 4117. Lecture Notes in Computer Science. Springer, 2006, pp. 142–159 (cit. on p. 30).

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. *Flush+Flush: A Fast and Stealthy Cache Attack*. In: *DIMVA*. Vol. 9721. Lecture Notes in Computer Science. Springer, 2016, pp. 279–299 (cit. on p. 27).

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. *Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks*. In: *USENIX Security Symposium*. USENIX Association, 2018, pp. 955–972 (cit. on p. 23).

[Gre18] Matthew Green. *Should you use SRP?* 2018. Link. (Cit. on p. 53).

[Gru14] B. Grubb. *Heartbleed Disclosure Timeline: Who Knew What and When*. Accessed: 2022-07-22. Apr. 2014. Link. (Cit. on pp. x, 3).

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. *Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches*. In: *USENIX Security Symposium*. USENIX Association, 2015, pp. 897–912 (cit. on p. 154).

[GZZY22] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang. *Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks*. In: *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1550–1550. Link. (Cit. on p. 28).

[Har08] Dan Harkins. *Simultaneous Authentication of Equals: A Secure, Password-Based Key Exchange for Mesh Networks*. In: *2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008)*. 2008, pp. 839–844 (cit. on pp. xii, 6, 31, 34).

[Har15] Dan Harkins. *"Dragonfly Key Exchange"*. In: *RFC* 7664 (2015), pp. 1–18 (cit. on pp. xiii, 7, 30, 34, 81, 84).

[Har19a] Dan Harkins. *Finding PWE in Constant Time*. July 2019. Link. (Cit. on p. 90).

[Har19b] Dan Harkins. *Improved Extensible Authentication Protocol Using Only a Password draft-harkins-eap-pwd-prime-00*. Internet-draft. July 2019. Link. (Cit. on pp. 35, 90).

[Har19c] Dan Harkins. *Secure Password Ciphersuites for Transport Layer Security (TLS)*. RFC 8492. Feb. 2019. Link. (Cit. on p. 34).

[HEC20] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. *ct-fuzz: Fuzzing for Timing Leaks*. In: *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 466–471. Link. (Cit. on pp. 105, 112, 113, 126).

[HGD+20] Sohaib ul Hassan, Iaroslav Gridin, Ignacio M. Delgado-Lozano, Cesar Pereida García, Jesús-Javier Chi-Domínguez, Alejandro Cabrera Aldaya, and Billy Bob Brumley. *Déjà Vu: Side-Channel Analysis of Mozilla's NSS*. In: *CCS*. ACM, 2020, pp. 1887–1902 (cit. on p. 46).

[HL19] Björn Haase and Benoît Labrique. *"AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT"*. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 1–48 (cit. on pp. 30, 31, 153).

[HO22] Feng Hao and Paul C. van Oorschot. *SoK: Password-Authenticated Key Exchange - Theory, Practice, Standardization and Real-World Lessons*. In: *AsiaCCS*. ACM, 2022, pp. 697–711 (cit. on pp. 30, 31).

[HR10] Feng Hao and Peter Ryan. *"J-PAKE: Authenticated Key Exchange without PKI"*. In: *Trans. Comput. Sci.* 11 (2010), pp. 192–206 (cit. on pp. 30, 31).

[HS14] Feng Hao and Siamak Fayyaz Shahandashti. *The SPEKE Protocol Revisited*. In: *SSR*. Vol. 8893. Lecture Notes in Computer Science. Springer, 2014, pp. 26–38 (cit. on pp. 31, 42).

[HTAP18] Julie M. Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. *"We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products*. In: *SOUPS @ USENIX Security Symposium*. USENIX Association, 2018, pp. 357–373 (cit. on p. 116).

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. *Practical Timing Side Channel Attacks against Kernel Space ASLR*. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 191–205 (cit. on pp. 22, 25).

[IEE12] IEEE. *"IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications"*. In: *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)* (2012), pp. 1–2793 (cit. on p. 34).

[IEE16] IEEE. *"IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications"*. In: *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (2016), pp. 1–3534 (cit. on pp. 34, 83, 84).

[IEE21] IEEE. *"IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks–Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications"*. In: *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)* (2021), pp. 1–4379 (cit. on pp. xiii, 7, 34–37, 95).

[IGI+16] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *Cache Attacks Enable Bulk Key Recovery on the Cloud*. In: *CHES*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 368–388 (cit. on p. 25).

[Ins20] Insomnia. *Insomnia Security Standards*. Accessed: 2022-07-22. 2020. Link. (Cit. on pp. 31, 53).

[Int21] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2021 (cit. on pp. 19, 20, 100).

[Jab96] David P. Jablon. *"Strong password-only authenticated key exchange"*. In: *Comput. Commun. Rev.* 26.5 (1996), pp. 5–26 (cit. on pp. 30, 31).

[Jab97] David P. Jablon. *Extended Password Key Exchange Protocols Immune to Dictionary Attacks*. In: *WETICE*. IEEE Computer Society, 1997, pp. 248–255 (cit. on p. 31).

[Jan21] Jan Jancar. *The state of tooling for verifying constant-timeness of cryptographic implementations*. 2021. Link. (Cit. on p. 109).

[Jas96] Barry Jaspan. *Dual-workfactor Encrypted Key Exchange: Efficiently Preventing Password Chaining and Dictionary Attacks*. In: *USENIX Security Symposium*. USENIX Association, 1996 (cit. on p. 42).

[JFB+22] J. Jancar, M. Fourné, D. De Almeida Braga, M. Sabt, P. Schwabe, G. Barthe, P. Fouque, and Y. Acar. *"They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks*. In: *2022 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 755–772. Link. (Cit. on pp. xv, xxi, 8, 154).

[JG04] Shaoquan Jiang and Guang Gong. *Password Based Key Exchange with Mutual Authentication*. In: *Selected Areas in Cryptography*. Vol. 3357. Lecture Notes in Computer Science. Springer, 2004, pp. 267–279 (cit. on p. 31).

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. *OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks*. In: *EUROCRYPT (3)*. Vol. 10822. Lecture Notes in Computer Science. Springer, 2018, pp. 456–486 (cit. on pp. 30, 31, 33, 153).

[JMH97] Teresa L. Johnson, Matthew C. Merten, and Wen-mei W. Hwu. *Run-Time Spatial Locality Detection and Optimization*. In: *MICRO*. ACM/IEEE Computer Society, 1997, pp. 57–64 (cit. on p. 19).

[Jou90] Norman P. Jouppi. *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers*. In: *ISCA*. ACM, 1990, pp. 364–373 (cit. on p. 19).

[JSSS20] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sýs. *"Minerva: The curse of ECDSA nonces; Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces"*. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.4 (2020), pp. 281–308. Link. (Cit. on p. 112).

[KDK+14] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. *Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors*. In: *ISCA*. IEEE Computer Society, 2014, pp. 361–372 (cit. on p. 46).

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. *Spectre Attacks: Exploiting Speculative Execution*. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019 (cit. on pp. 46, 110, 114).

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. *Differential Power Analysis*. In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397. Link. (Cit. on pp. x, 4, 46).

[KMO12] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. *Automatic Quantification of Cache Side-Channels*. In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. LNCS. Springer, 2012, pp. 564–580. Link. (Cit. on pp. 112, 113, 126).

[KNR+17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. *CogniCrypt: supporting developers in using cryptography*. In: *ASE*. IEEE Computer Society, 2017, pp. 931–936 (cit. on p. 116).

[Koc96] Paul C. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. In: *Advances in Cryptology – CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. LNCS. Springer, 1996, pp. 104–113. Link. (Cit. on pp. x, 4, 109, 111).

[KPVV16] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. *When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015*. In: *Cryptology and Network Security*. Ed. by Sara Foresti and Giuseppe Persiano. Vol. 10052. LNCS. Springer, 2016, pp. 573–582. Link. (Cit. on p. 111).

[KSA+21] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. *"CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs"*. In: *IEEE Trans. Software Eng.* 47.11 (2021), pp. 2382–2400 (cit. on p. 116).

[Kwo01] Taekyoung Kwon. *Authentication and Key Agreement Via Memorable Passwords*. In: *NDSS*. The Internet Society, 2001 (cit. on p. 31).

[Kwo05] Taekyoung Kwon. *"Revision of AMP in IEEE P1363.2 and ISO/IEC 11770-4"*. In: *Submission to IEEE P1363, 2005* (2005) (cit. on p. 31).

[Lan10] Adam Langley. *ctgrind*. 2010. Link. (Cit. on pp. xvi, 9, 105, 109, 112, 113, 126).

[Ler09] Xavier Leroy. *"Formal verification of a realistic compiler"*. In: *Commun. ACM* 52.7 (2009), pp. 107–115 (cit. on p. 145).

[LGR21] Julia Len, Paul Grubbs, and Thomas Ristenpart. *Partitioning Oracle Attacks*. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 195–212. Link. (Cit. on p. 42).

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. *ARMageddon: Cache Attacks on Mobile Devices*. In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 549–564 (cit. on pp. 47, 71).

[LKO+21] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. *PLATYPUS: Software-based Power Side-Channel Attacks on x86*. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021 (cit. on p. 46).

[LMD+22] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. *"DRAWNAPART: A Device Identification Technique based on Remote GPU Fingerprinting"*. In: *CoRR* abs/2201.09956 (2022) (cit. on p. 23).

[Log17] LogMeIn. *Remote Support and Service Desk Security*. 2017. Link. (Cit. on p. 72).

[Log20] LogMeIn. *Web conference security*. 2020. Link. (Cit. on p. 72).

[LS15] Jean Lancrenon and Marjan Skrobot. *On the Provable Security of the Dragonfly Protocol*. In: *ISC*. Vol. 9290. Lecture Notes in Computer Science. Springer, 2015, pp. 244–261 (cit. on p. 34).

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. *Meltdown: Reading Kernel Memory from User Space*. In: *USENIX Security Symposium*. USENIX Association, 2018, pp. 973–990 (cit. on p. 46).

[LSR+22] Emiliano Lorini, Nicolas Sabouret, Brian Ravenet, Jorge Fernandez, and Céline Clavel. *Cognitive Planning in Motivational Interviewing*. SCITEPRESS, 2022, pp. 508–517 (cit. on p. 117).

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. *Last-Level Cache Side-Channel Attacks are Practical*. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 605–622 (cit. on pp. 24, 25).

[LZJZ21] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. *"A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography"*. In: *ACM Comput. Surv.* 54.6 (2021), 122:1–122:37 (cit. on pp. xi, 5).

[Mac01] Philip MacKenzie. *"On the Security of the SPEKE Password-Authenticated Key Exchange Protocol"*. In: *IACR Cryptol. ePrint Arch.* (2001), p. 57 (cit. on pp. 31, 42).

[MBA+21] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. *Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)*. In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 213–230 (cit. on p. 46).

[MKJR16] Kathleen M. Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. *"PKCS #1: RSA Cryptography Specifications Version 2.2"*. In: *RFC* 8017 (2016), pp. 1–78 (cit. on pp. xvii, 10).

[Mon85] P. Montgomery. *"Modular multiplication without trial division"*. In: *Mathematics of Computation* 44 (1985), pp. 519–521 (cit. on p. 55).

[MS14] Daniel Mayer and Joel Sandin. *Time Trial: Racing Towards Practical Remote Timing Attacks*. Tech. rep. NCC Group, 2014. Link. (Cit. on p. 112).

[MSF19] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. *"Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice"*. In: *Proc. ACM Hum. Comput. Interact.* 3.CSCW (2019), 72:1–72:23 (cit. on p. 119).

[MUTS22] Soundes Marzougui, Vincent Ulitzsch, Mehdi Tibouchi, and Jean-Pierre Seifert. *"Profiling Side-Channel Attacks on Dilithium: A Small Bit-Fiddling Leak Breaks It All"*. In: *IACR Cryptol. ePrint Arch.* (2022), p. 106 (cit. on p. 112).

[MWS+17] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. *Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud*. In: *NDSS*. The Internet Society, 2017 (cit. on p. 23).

[Nei] Moritz Neikes. *TIMECOP*. Link. (Cit. on pp. xvi, 9, 91, 105, 112, 113, 126, 139).

[Nik09] Cubrilovic Nik. *RockYou Hack: From Bad To Worse*. Accessed: 2022-07-22. Dec. 2009. Link. (Cit. on pp. 66, 84).

[Nik19] Nikolai Tschacher. *Model Based fuzzing of the WPA3 Dragonfly Handshake*. MA thesis. Berlin, Germany: Institute for Computer Science, Humboldt University, 2019 (cit. on p. 86).

[NIS16] NIST. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. 2016. Link. (Cit. on p. 109).

[NKS+17a] Matús Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. *Measuring Popularity of Cryptographic Libraries in Internet-Wide Scans*. In: *ACSAC*. ACM, 2017, pp. 162–175 (cit. on p. 68).

[NKS+17b] Matús Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. *Measuring Popularity of Cryptographic Libraries in Internet-Wide Scans*. In: *ACSAC*. ACM, 2017, pp. 162–175 (cit. on p. 114).

[NS06] Michael Neve and Jean-Pierre Seifert. *Advances on Access-Driven Cache Attacks on AES*. In: *Selected Areas in Cryptography*. Vol. 4356. Lecture Notes in Computer Science. Springer, 2006, pp. 147–162 (cit. on p. 25).

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. *The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications*. In: *CCS*. ACM, 2015, pp. 1406–1418 (cit. on pp. 24, 47).

[OSC19] Open System Consultants (OSC). *Two vulnerabilities in Radiator: EAP-pwd authentication bypass and DoS with certain TLS configurations*. Accessed: 2020-09-03. Apr. 2019. Link. (Cit. on p. 81).

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. *Cache Attacks and Countermeasures: The Case of AES*. In: *CT-RSA*. Vol. 3860. Lecture Notes in Computer Science. Springer, 2006, pp. 1–20 (cit. on pp. 24, 25, 111).

[PBP+20] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella Béguelin. *HACLxN: Verified Generic SIMD Crypto (for all your favourite platforms)*. In: *CCS*. ACM, 2020, pp. 899–918 (cit. on pp. 114, 139, 140).

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. *To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures*. In: *CCS'17*. ACM, 2017, pp. 1843–1855 (cit. on pp. 27, 112).

[Per05] Colin Percival. *Cache missing for fun and profit*. 2005 (cit. on p. 54).

[Per13] Trevor Perrin. *[TLS] Question regarding CFRG process*. Dec. 2013. Link. (Cit. on pp. 34, 79).

[Per22] Cesar Pereida Garcia. *Side-Channel Analysis and Cryptography Engineering : Getting OpenSSL Closer to Constant-Time*. PhD Thesis. Tampere University, 2022. Link. (Cit. on p. 149).

[PK94] Subbarao Palacharla and Richard E. Kessler. *Evaluating Stream Buffers as a Secondary Cache Replacement*. In: *ISCA*. IEEE Computer Society, 1994, pp. 24–33 (cit. on p. 19).

[PLF21] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. *Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical*. In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 645–662 (cit. on p. 23).

[PPF+20] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 983–1002 (cit. on pp. 114, 139, 140).

[PT19] Thales Bandiera Paiva and Routo Terada. *A Timing Attack on the HQC Encryption Scheme*. In: *Selected Areas in Cryptography – SAC 2019*. Ed. by Kenneth G. Paterson and Douglas Stebila. Vol. 11959. LNCS. Springer, 2019, pp. 551–573. Link. (Cit. on p. 112).

[PTV21] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. *Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks*. In: *CCS*. ACM, 2021, pp. 2906–2920 (cit. on p. 25).

[PW17] David Pointcheval and Guilin Wang. *VTBPEKE: Verifier-based Two-Basis Password Exponential Key Exchange*. In: *AsiaCCS*. ACM, 2017, pp. 301–312 (cit. on p. 31).

[PZR+17] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. *"Verified low-level programming embedded in F"*. In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 17:1–17:29 (cit. on p. 141).

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. *Dude, is my code constant time?* In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 1697–1702. Link. (Cit. on pp. 112, 113, 126).

[RCF+20] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. *"Towards making formal methods normal: meeting developers where they are"*. In: *CoRR* abs/2010.16345 (2020) (cit. on pp. 116, 134).

[RMBO22] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. *Port Contention Goes Portable: Port Contention Side Channels in Web Browsers*. In: *AsiaCCS*. ACM, 2022, pp. 1182–1194 (cit. on p. 23).

[RPA16] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. *Sparse representation of implicit flows with applications to side-channel detection*. In: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. Ed. by Ayal Zaks and Manuel V. Hermenegildo. ACM, 2016, pp. 110–120. Link. (Cit. on pp. 112, 113, 126).

[RR01] Josyula R. Rao and Pankaj Rohatgi. *"EMpowering Side-Channel Attacks"*. In: *IACR Cryptol. ePrint Arch.* (2001), p. 37 (cit. on pp. x, 4).

[Rus21] Andy Russon. *Threat for the Secure Remote Password Protocol and a Leak in Apple's Cryptographic Library*. In: *ACNS (2)*. Vol. 12727. Lecture Notes in Computer Science. Springer, 2021, pp. 49–75 (cit. on pp. 43, 67).

[SAO+21] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. *Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses*. In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 2863–2880 (cit. on p. 23).

[SAOP17] Stanislav Smyshlyaev, Evgeny K. Alekseev, Igor B. Oshkin, and Vladimir Popov. *"The Security Evaluated Standardized Password-Authenticated Key Exchange (SESPAKE) Protocol"*. In: *RFC* 8133 (2017), pp. 1–51 (cit. on p. 31).

[SBB+22] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. *"Spectre Declassified: Reading from the Right Place at the Wrong Time"*. In: *IACR Cryptol. ePrint Arch.* (2022), p. 426 (cit. on p. 145).

[Sch17] Jörn-Marc Schmidt. *"Requirements for Password-Authenticated Key Agreement (PAKE) Schemes"*. In: *RFC* 8125 (2017), pp. 1–10 (cit. on pp. xiii, 6, 31).

[Sec] Defuse Security. *CrackStation's Password Cracking Dictionary (Human Passwords Only*. Link. (Cit. on p. 84).

[Sha48] Claude E. Shannon. *"A mathematical theory of communication"*. In: *Bell Syst. Tech. J.* 27.3 (1948), pp. 379–423 (cit. on p. 41).

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. *Dependent types and multi-monadic effects in F*. In: *POPL*. ACM, 2016, pp. 256–270 (cit. on p. 141).

[Sho20] Victor Shoup. *"Security analysis of SPAKE2+"*. In: *IACR Cryptol. ePrint Arch.* (2020), p. 313 (cit. on p. 31).

[SK12] SeongHan Shin and Kazukuni Kobara. *"Efficient Augmented Password-Only Authentication and Key Exchange for IKEv2"*. In: *RFC* 6628 (2012), pp. 1–20 (cit. on p. 31).

[SL19] Wei Song and Peng Liu. *Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC*. In: *RAID*. USENIX Association, 2019, pp. 427–442 (cit. on p. 25).

[SLL+20] Alan T. Sherman, Erin Lanus, Moses Liskov, Edward Zieglar, Richard Chang, Enis Golaszewski, Ryan Wnuk-Fink, Cyrus J. Bonyadi, Mario Yaksetig, and Ian Blumenfeld. *Formal Methods Analysis of the Secure Remote Password Protocol*. In: *Logic, Language, and Security*. Vol. 12300. Lecture Notes in Computer Science. Springer, 2020, pp. 103–126 (cit. on p. 42).

[Smi82] Alan Jay Smith. *"Cache Memories"*. In: *ACM Comput. Surv.* 14.3 (1982), pp. 473–530 (cit. on p. 19).

[ST16] Mohamed Sabt and Jacques Traoré. *Cryptanalysis of globalplatform secure channel protocols*. In: *International Conference on Research in Security Standardisation*. Springer. 2016, pp. 62–91 (cit. on pp. xvii, 9).

[SVWW21] Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. *A Coq proof of the correctness of X25519 in TweetNaCl*. In: *IEEE Computer Security Foundations Symposium (CSF)*. 2021, pp. 1–16 (cit. on p. 140).

[Swa22] Nikhil Swamy. *Proof-Oriented Programming in F**. 2022. Link. (Cit. on p. 141).

[Tea20] The Clang Team. *MemorySanitizer*. Accessed: 2022-07-22. 2020. Link. (Cit. on pp. 112, 113, 126).

[Tec18] GlobalPlatform Technology. *Card Specification Version 2.3.1*. Mar. 2018 (cit. on pp. xvi, 9).

[Tec20] GlobalPlatform Technology. *Secure Channel Protocol '10' – Card Specification v2.3 – Amendment l*. 2020 (cit. on pp. xvii, 10).

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*. Ed. by David A. Patterson. ACM, 1995, pp. 392–403. Link. (Cit. on p. 18).

[Tom16] Aaron Tomb. *"Automated Verification of Real-World Cryptographic Implementations"*. In: *IEEE Secur. Priv.* 14.6 (2016), pp. 26–33 (cit. on p. 140).

[Tom21] David Tomaschik. *GPU Accelerated Password Cracking in the Cloud: Speed and Cost-Effectiveness*. Accessed: 2022-07-22. 2021. Link. (Cit. on p. 67).

[TSS+03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. *Cryptanalysis of DES implemented on computers with cache*. In: *Cryptographic Hardware and Embedded Systems – CHES 2003*. Vol. 2779. LNCS. Springer, 2003, pp. 62–76 (cit. on p. 111).

[TTMM02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. *Cryptanalysis of Block Ciphers Implemented on Computers with Cache*. In: *Proceedings of the International Symposium on Information Theory and Its Applications, ISITA 2002*. 2002, pp. 803–806 (cit. on p. 111).

[TW21] Mehdi Tibouchi and Alexandre Wallet. *"One Bit is All It Takes: A Devastating Timing Attack on BLISS's Non-Constant Time Sign Flips"*. In: *J. Math. Cryptol.* 15.1 (2021), pp. 131–142 (cit. on p. 112).

[TWMP07] David Taylor, Thomas Wu, Nikos Mavrogiannopoulos, and Trevor Perrin. *"Using the Secure Remote Password (SRP) Protocol for TLS Authentication"*. In: *RFC* 5054 (2007), pp. 1–24 (cit. on pp. 54, 64, 69, 71).

[UCSa] UCSD PLSysSec. *haybale-pitchfork*. Link. (Cit. on pp. 112, 113, 126).

[UCSb] UCSD PLSysSec. *pitchfork-angr*. Link. (Cit. on p. 114).

[Van22] Mathy Vanhoef. *A Time-Memory Trade-Off Attack on WPA3's SAE-PK*. In: *APKC@AsiaCCS*. ACM, 2022, pp. 27–37 (cit. on p. 44).

[VKM19] Pepe Vila, Boris Köpf, and José F. Morales. *Theory and Practice of Finding Eviction Sets*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 39–54 (cit. on p. 25).

[Vol16] Ronald Volgers. *Exploiting two buggy SRP implementations*. 2016. Link. (Cit. on p. 43).

[VP16] Mathy Vanhoef and Frank Piessens. *Predicting, Decrypting, and Abusing WPA2/802.11 Group Keys*. In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 673–688 (cit. on p. 79).

[VP17] Mathy Vanhoef and Frank Piessens. *Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2*. In: *CCS*. ACM, 2017, pp. 1313–1328 (cit. on pp. xiii, 7, 79).

[VP18] Mathy Vanhoef and Frank Piessens. *Release the Kraken: New KRACKs in the 802.11 Standard*. In: *CCS*. ACM, 2018, pp. 299–314 (cit. on p. 79).

[VR20] Mathy Vanhoef and Eyal Ronen. *Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd*. In: *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 517–533 (cit. on pp. xiii, xiv, 7, 35, 40, 42, 43, 67, 79, 81, 86, 90, 103, 104).

[WBB+20] Guillaume Wafo-Tapa, Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Etienne Marcatel. *"A practicable timing attack against HQC and its countermeasure"*. In: *Advances in Mathematics of Computation* (2020). Link. (Cit. on p. 112).

[WGSW18] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. *Eliminating timing side-channel leaks using program repair*. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 15–26. Link. (Cit. on pp. 113, 126).

[WMES18a] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. *MicroWalk: A Framework for Finding Side Channels in Binaries*. In: *ACSAC*. ACM, 2018, pp. 161–173 (cit. on pp. xiv, 7, 90).

[WMES18b] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. *MicroWalk: A Framework for Finding Side Channels in Binaries*. In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 161–173. Link. (Cit. on pp. 112, 113, 126).

[WPH+22] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. *Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86*. In: *Proceedings of the USENIX Security Symposium (USENIX)*. 2022 (cit. on p. 23).

[WRP+19] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. *"CT-wasm: type-driven secure cryptography for the web ecosystem"*. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 77:1–77:29. Link. (Cit. on pp. 112, 113, 126).

[WSBS20] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. *Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations*. In: *USENIX Security Symposium*. USENIX Association, 2020, pp. 1767–1784 (cit. on pp. 112, 113, 125, 126, 139).

[Wu00] Thomas Wu. *"Telnet Authentication: SRP"*. In: *RFC* 2944 (2000), pp. 1–7 (cit. on p. 31).

[Wu02] Thomas Wu. *"SRP-6: Improvements and Refinements to the Secure Remote Password Protocol"*. In: *Submission to the IEEE P1363 Working Group* (2002) (cit. on pp. 31, 42).

[Wu09] Thomas Wu. *SRP Protocol Design (SRP-6a)*. 2009. Link. (Cit. on pp. xii, 6, 31).

[Wu98] Thomas D. Wu. *The Secure Remote Password Protocol*. In: *NDSS*. The Internet Society, 1998 (cit. on pp. 30, 31).

[WWL+17] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. *CacheD: Identifying Cache-Based Timing Channels in Production Software*. In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 235–252. Link. (Cit. on pp. 112, 113, 126).

[WZS+18] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. *DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries*. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 603–620. Link. (Cit. on pp. 112, 113, 126).

[Yar16] Yuval Yarom. *Mastik: A Micro-Architectural Side-Channel Toolkit*. 2016. Link. (Cit. on pp. 64, 86, 98).

[YB14] Yuval Yarom and Naomi Benger. *"Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack"*. In: *IACR Cryptol. ePrint Arch.* (2014), p. 140 (cit. on p. 27).

[YF14] Yuval Yarom and Katrina Falkner. *FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack*. In: *USENIX Security Symposium*. USENIX Association, 2014, pp. 719–732 (cit. on pp. 25, 27, 47).

[YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. *CacheBleed: A Timing Attack on OpenSSL Constant Time RSA*. In: *CHES*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 346–367 (cit. on p. 111).

[ZBPB17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. *HACL*: A Verified Modern Cryptographic Library*. In: *CCS*. ACM, 2017, pp. 1789–1806 (cit. on pp. xvi, 9, 114, 139, 140).

[ZH10] Glen Zorn and Dan Harkins. *Extensible Authentication Protocol (EAP) Authentication Using Only a Password*. RFC 5931. Aug. 2010. Link. (Cit. on pp. 34, 38, 84).

[Zha04] Muxiang Zhang. *"Analysis of the SPEKE password-authenticated key exchange protocol"*. In: *IEEE Commun. Lett.* 8.1 (2004), pp. 63–65 (cit. on p. 42).

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. *Cross-VM side channels and their use to extract private keys*. In: *the ACM Conference on Computer and Communications Security, CCS'12*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM, 2012, pp. 305–316. Link. (Cit. on pp. x, 4).

Appendix

Chosen Group Attack Against ProtonMail' SRP

In this appendix, we detail the vulnerability allowing attackers to perform password recovery attacks against ProtonMail's authentication with its python client.

First, we recall that ProtonMail aims at providing mail confidentiality even when the server is malicious. Hence, we can consider an attacker controlling the server's responses.

Upon authentication, a user must run an Secure Remote Password protocol (SRP) session using its password to negotiate its authentication token. When authentication using their python client, the script relies on two main modules: pySRP (embedded in the project) and python-gnupg.

We note that pySRP has a very flexible API, allowing the use of non-standard group parameters to be used. The server provides these parameters at the start of the session, signed using its PGP key (the public key is hardcoded). We note that the generator g is fixed to the value 2.

If the server is malicious, it can choose weak parameters. Otherwise, if we assume an attacker in a Machine-in-the-Middle (MitM) position, they would need to bypass the signature. Luckily, the signature is verified using python-gnupg, enabling such an attack. In fact, despite the ssh public key of the server being hardcoded in the python client, python-gnupg initialize the created key ring with the user's system keyring, thereby importing all public keys the user has added. Then, the signature verification is successful if *any public key* can be used to verify it. This means any attacker who successfully made the user import their public key can bypass the signature verification and choose the group parameters.

Chosen Group Attack

Considering SRP's workflow (see section 3.2), the attacker cannot learn $g^x \bmod p$. Hence, the best strategy would be to choose group parameters to recover the ephemeral key share from the user and perform an offline dictionary attack from a single session. The attack would go as follow, with chosen values displayed in red and public values in blue:

1. Client starts the session by sending its id to the server
2. The attacker intercepts the response and replaces the group prime modulus p by a composite integer n, the product of small prime factors. This means the attacker can easily solve the Discrete Logarithm Problem (DLP) over the negotiated group. The attacker may also choose the value of B to make the remaining attack easier. They transmit (salt, n) to the victim.
3. The victim privately computes the following:
   - $x = H(id,\ H(\text{pwd} : salt))$

- $A = 2^a \bmod n$
- $S = (B - 2^x)^{(a+ux)} \bmod n$

In a normal run, the victim would compute K = H(S), but ProtonMail uses K = S. The victim sends (A, M = H(A || B || S))

4. The attacker solve the DLP on A and recovers a. Then, they can test every password from a dictionary by computing the values x' and S'. If H(A || B || S') == M, then the password is valid.

Considering the recovered information, the attacker would recover enough information from a single remote session to recover the password from any reasonable dictionary.

Additional material regarding Dragondoom

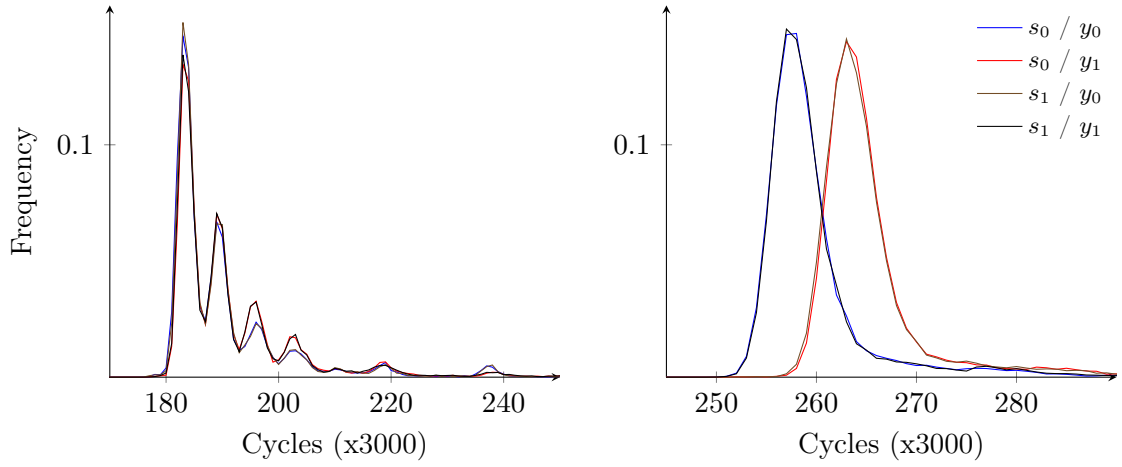
B

B.1. Results of the PDA on Affected Libraries

For all affected libraries, we perform a Performance Degradation Attack (PDA) on the conditional instructions, while varying the parity of the seed (used as the compression format) and the y-coordinate obtained from Tonelli-Shanks.

Results for OpenSSL, WolfSSL and ell are depicted in Figure B.1, Figure B.2 and Figure B.3 respectivelly. For each figure, we display a reference graph, representing the average number of cycles for an execution where no PDA is applied. Then, we applied the PDA at relevant places to highlight the impact on the number of cycles needed to perform the point decompression. In the legend of each graph, s represents the seed, and y the coordinate obtained from Tonelli-Shanks; the index is the parity of the corresponding value. For instance, "s_0 / y_0" means that both the seed and the coordinate are even.

For OpenSSL, we notice two distributions in Figure B.1b: (i) the curves "s_0 / y_0" and "s_1 / y_1" share the same distribution; (ii) the curves "s_0 / y_1" and "s_1 / y_0" share another. This suggests that attackers can learn whether the parity of the seed and the coordinate are the same.

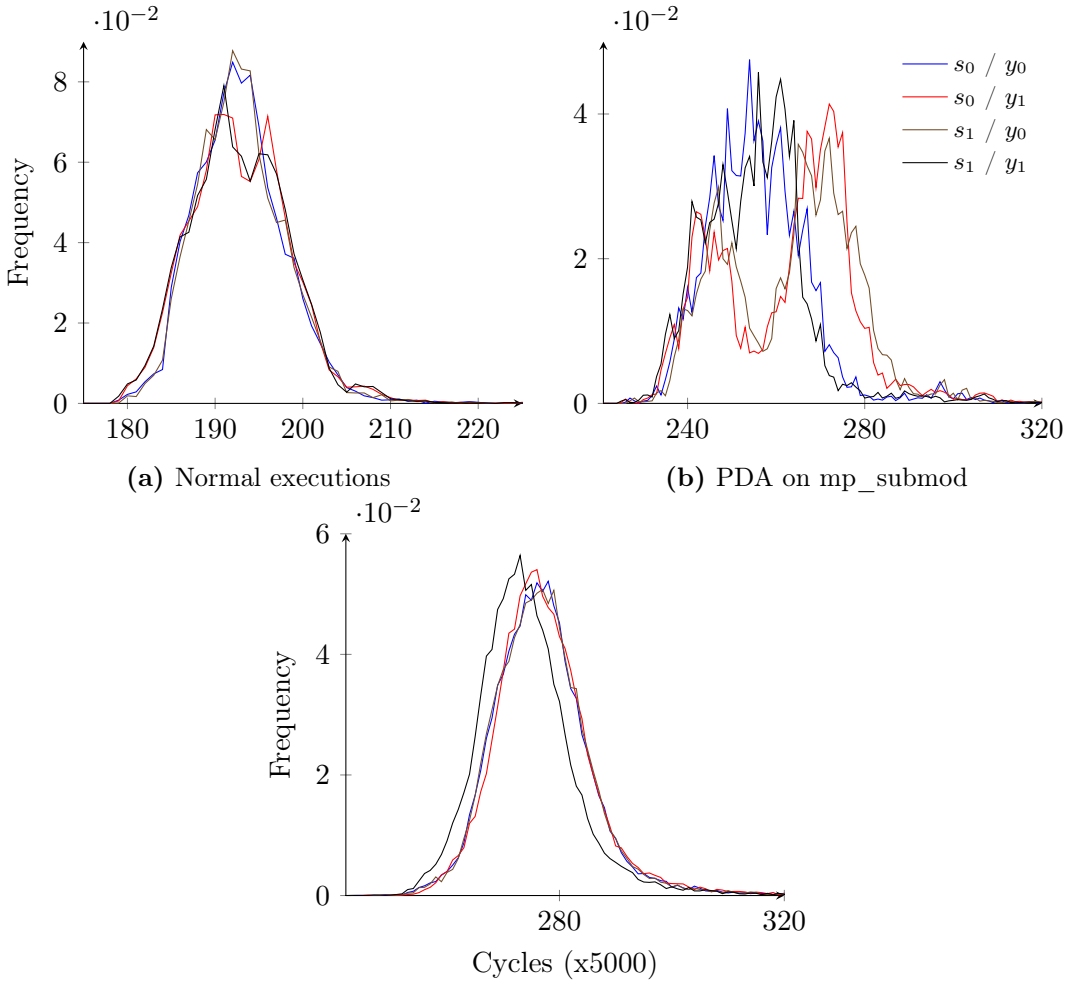


(a) Normal executions

(b) PDA on the condition

Figure B.1. – Number of cycles per execution of the point decompression in OpenSSL when the seed and y parity vary.

For WolfSSL, we observe different results depending on where we perform the PDA. If we do it on the conditional subtraction, we can distinguish the curves "s_0 / y_0" and "s_1 / y_1" from the two other curves, as for OpenSSL. Furthermore, applying the PDA on the parity verification induce a small difference between the case "s_1 / y_1" and all others. Hence, attackers may get a little more information, although the leakage is mild.



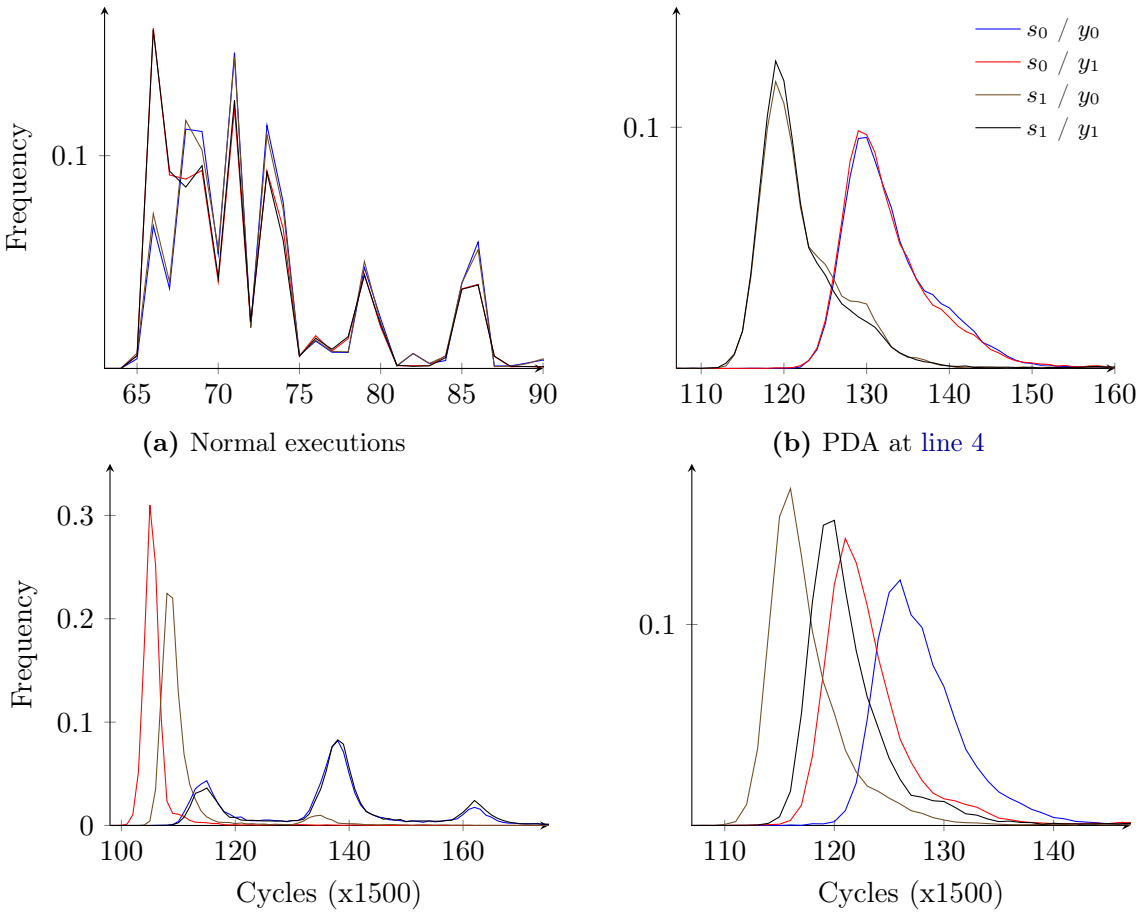
(a) Normal executions

(b) PDA on mp_submod

(c) PDA on mp_isodd

Figure B.2. – Number of cycles per execution of the point decompression in WolfSSL when the seed and y parity vary.

For ell, we achieve a clearer result. Figure B.3b shows a clear distinction between the curves "s_0 / y_1" and "s_0 / y_0" and the two others, leaking the parity of the seed. Figure B.3c, in addition, shows a clear distinction between curves "s_0 / y_0" and "s_1 / y_1" (seed and y have the same parity) and the two others, with a slight difference between "s_0 / y_1" and "s_1 / y_0". Finally, combining the PDA at two places, as depicted in Figure B.3d gives four distinct distributions, suggesting attackers would be able to distinguish the four cases.



(a) Normal executions

(b) PDA at line 4

(c) PDA at line 6

(d) PDA at line 4 and line 6

Figure B.3. – Number of cycles per execution of the point decompression in ell when the seed and y parity vary. Line numbers in captions refer to Listing 7.8.

B.2. Experiments on BN_bin2bn

The second vulnerability may be exploited using the same Flush+Reload-gadget described in subsection 7.3.3, using the number of cache-hits as an indicator to learn how many leading bytes are skipped.

The leakage results in a smaller instruction discrepancy, making it more difficult to reliably exploit. In our preliminary testing, we repeated measurements 60 times for each value to make the leakage more apparent, but less repetitions may be sufficient to achieve an equivalent performance.

Figure B.4 represents the average number of hits per measurement, with an increasing number of leading bytes to zero. While attackers may not reliably guess the exact number of Most Significant Bit (MSB) to zero, they can clearly identify whether the value has at least one or not.

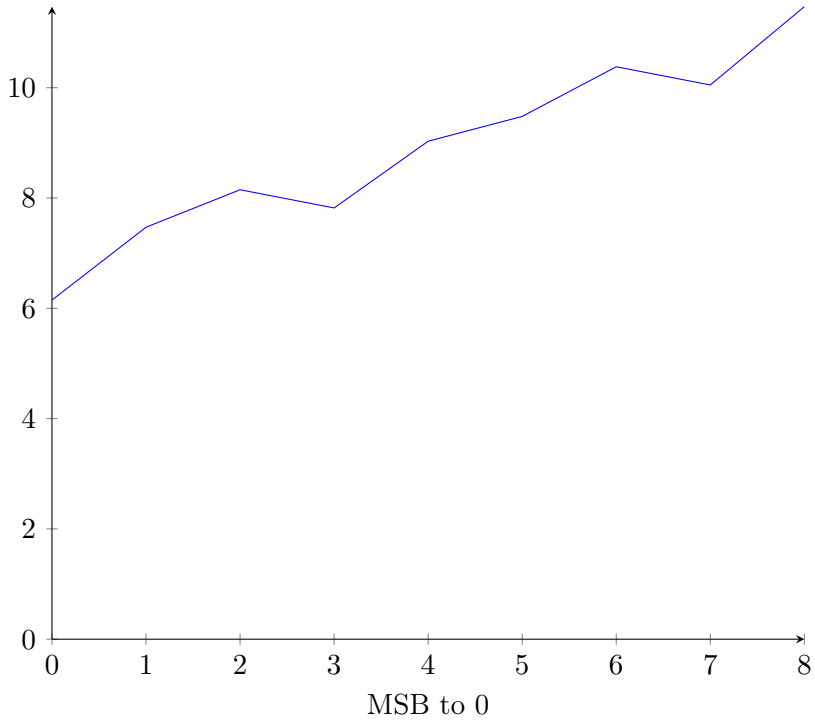


Figure B.4. – Average cache-hits in BN_bin2bn for an increasing number of leading zero bytes.

C Survey

C.1. Background

Q1.1: How many years have you been developing cryptographic code?
[Numeric field]
Q1.2: What background do you have in cryptography?

- ☐ Academic
- ☐ Took some classes
- ☐ On the job experience
- ☐ Teach it
- ☐ Hobby
- ☐ Industry
- ☐ Prefer not to say

Q1.3: Can you tell us a little bit more about your background as a developer who works on cryptographic libraries/primitives?
[Free text field]

C.2. Library / Primitive

Q2.1: What's your role in the development of *library*? (*e.g.,* maintainer, project lead, core developer, commit rights, no rights, etc.)
[Free text field]
Q2.2: How are you involved in design decisions (*e.g.,* concerning the API, coding guidelines and style, security-relevant properties) for *library*?
[Free text field]
Q2.3: What are the intended use cases of *library*? (*e.g.,* embedded use, servers, etc.)
[Free text field]
Q2.4: What is the threat model for *library* with regards to side-channel attacks? (*e.g.,* local/remote attackers, etc.)
[Free text field]
Q2.5: Do you consider timing attacks a relevant threat for the intended use of *library* and its threat model? Please give a brief explanation for why / why not. (If the execution time of a program depends on secret data, a timing attack recovers information about the secret by computing the inverse of this dependency. The two most notorious sources for such dependencies are secret dependent control flow and secret-dependent memory access. Timing attacks include cache attacks where the attacker uses the cache to infer information about memory accesses of a target.)
[Free text field]
Q2.6: Does *library* claim resistance against timing attacks?

- ○ Yes
- ○ No
- ○ Partially
- ○ I don't know
- ○ Not yet but planning to

Q2.7: How did the development team decide to protect or not to protect against timing attacks? (We are interested in the decision process and not the protection mechanisms themselves (if any).)
[Free text field]
Q2.8: [only shown if **Q2.6** is "Yes" or "Partially"] How does *library* protect against timing attacks?
[Free text field]
Q2.9: Did you personally test for or verify the resistance of *library* against timing attacks?

- ○ Yes
- ○ No
- ○ Partially
- ○ Not yet but planning to
- ○ Not me but someone did
- ○ I don't know
- ○ Prefer not to say

Q2.10: [only shown if **Q2.9** is "Yes" or "Partially"] How did you test or verify the resistance against timing attacks? (*e.g.,* using which tools, techniques, practices.)
[Free text field]
Q2.11: [only shown if **Q2.9** is "Yes" or "Partially"] How often do you test or verify the resistance of *library* against timing attacks?

- ☐ Only did it once
- ☐ Do it occasionally
- ☐ During releases
- ☐ During CI
- ☐ Don't know
- ☐ Prefer not to say

C.2.1. Tooling

Q3.1: Are you aware of tools that can test or verify resistance against timing attacks?

- ○ Yes
- ○ No

Q3.2: Please tell us which of these you've heard of with regards to verifying resistance against timing attacks.
[List of tools from Table 8.1.]
Q3.3: How did you learn about them? (Check all that apply)
[Matrix question with subquestions being the tools the participant selected in **Q3.2** and the following answer options:]

- ☐ Recommended by colleague
- ☐ Heard from authors
- ☐ Read the paper

- ☐ Referenced in a blog/different paper
- ☐ Was involved in the development
- ☐ Other

Q3.4: Which of these (if any) have you tried to use in the context of resistance against timing attacks?
[Multiple choice question among the tools selected by the participant in **Q3.2**.]
Q3.5: Why have you not tried to use these?
[Multiple free text fields for all of the tools the participant did select in **Q3.2** but not in **Q3.4**.]

C.3. Tool use

[All of the questions in this group are matrix questions with subquestions for all of the tools the participant did select in **Q3.2** and **Q3.4**, *i.e.,* those tools that the participant knows and tried to use.]
Q4.1: Please describe the process of using the tools.
[Free text field]
Q4.2: I was satisfied with the installation process. (Please rate your agreement with the above statement.)

- ○ I quit using the tool before I got to this point
- ○ I quit using the tool because this was a problem
- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree or disagree
- ○ Agree
- ○ Strongly agree

Q4.3: I was satisfied with the prerequisites that the tool needed to work with my code. (Please rate your agreement with the above statement.)
[Same answer options as **Q4.2**]
Q4.4: In my understanding the tool is sound. (Please rate your agreement with the above statement. A sound tool only deems secure programs secure, thus has no false negatives.)
[Same answer options as **Q4.2**]
Q4.5: In my understanding the tool is complete. (Please rate your agreement with the above statement. A complete tool only deems insecure programs insecure, thus has no false positives.)
[Same answer options as **Q4.2**]
Q4.6: I understood the results the tool provided. (Please rate your agreement with the above statement.)
[Same answer options as **Q4.2**]

Q4.7: I was satisfied with the documentation of the tool. (Please rate your agreement with the above statement.)

[Same answer options as **Q4.2**]

Q4.8: I was satisfied with the overall usability of the tool. (Please rate your agreement with the above statement.)

[Same answer options as **Q4.2**]

Q4.9: I was satisfied with the tool overall. (Please rate your agreement with the above statement.)

[Same answer options as **Q4.2**]

C.4. Tool use: Dynamic instrumentation based

Q5.1: Use of dynamic instrumentation based tools like ctgrind, MemSan or Timecop requires:

- Creating test harnesses.
- Annotating secret inputs in the code.
- Compiling code with a specific compiler (in the MemSan case).

and in return detects non-constant time code that was executed (*e.g.,* branches on secret values, or secret-dependent memory accesses). However, it does not detect non-constant time code that was not executed (in branches not executed due conditions on public inputs).

Do you think you would fulfill these requirements in order to use this type of tool?

[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]

Q5.2: Can you clarify your reasoning for the answer?

- ☐ Not my decision
- ☐ Not applicable to my library
- ☐ Would like the guarantees but too much effort
- ☐ Good tradeoff of requirements and guarantees
- ☐ Already using one of the mentioned tools
- ☐ Will try to use one of the mentioned tools after this survey
- ☐ I don't care about the guarantees
- ☐ None of the above

Q5.3: Please expand on your answer if the above question didn't suffice?

[Free text field]

C.5. Tool use: Statistical runtime tests

Q6.1: Use of runtime statistical test-based tools like dudect requires:

- Creating a test harness that creates a list of public inputs and a list of representatives of two classes of secret inputs for which runtime variation will be tested.

and in return provides statistical guarantees of constant-timeness obtained by running the target code many times and performing statistical analysis of the results.

Do you think you would fulfill these requirements in order to use this type of tool?

[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]

Q6.2: Can you clarify your reasoning for the answer?

[Same answer options as **Q5.2**]

Q6.3: Please expand on your answer if the above question didn't suffice?

[Free text field]

C.6. Tool use: Formal analysis

Q7.1: Use of formal analysis-based tools like ct-verif requires:

- Annotation of the secret and public inputs in the source code.
- Running the analysis via a formal verification toolchain (*i.e.,* SMACK).
- Might not handle arbitrarily large programs or might require assistance in annotation of loop bounds.

and in return provides sound and complete guarantees (no false positives or negatives) of constant-timeness (*e.g.,* no branches on secrets or secret-dependent memory accesses or secret inputs to certain instructions).

Do you think you would fulfill these requirements in order to use this type of tool?

[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]

Q7.2: Can you clarify your reasoning for the answer?

[Same answer options as **Q5.2**]

Q7.3: Please expand on your answer if the above question didn't suffice?

[Free text field]

C.7. Miscellaneous

Q8.1: Do you have any other thoughts on timing attacks that you want to share?

[Free text field]

Q8.2: Do you have any other thoughts on or experiences with those tools that you want to share?

[Free text field]

Q8.3: Do you have any feedback on this survey, research, or someone you think we should talk to about this research (ideally an email address we could reach)?

[Free text field]

Q8.4: Do you want to allow us to contact you for:

- ☐ sending you a report of our results from the survey
- ☐ asking possible follow-up questions

Q8.5: [Only shown if some of the options in **Q8.4** was selected] To allow us to contact you, please enter your preferred email address. (If at any time you want to revoke consent to contact you and ask us to delete your email address, please email [de-identified for submission])
[Free text field]

UNIVERSITÉ DE RENNES 1

Titre : Cryptographie dans la Nature : La Sécurité des Implémentations Cryptographiques

Mot clés : PAKE, Attaque sur le cache, Dragonfly, SRP

Résumé : Les attaques par canaux auxiliaire sont redoutables face aux implémentations cryptographiques. Malgré les attaques passées, et la prolifération d'outils de vérification, ces attaques affectent encore de nombreuses implémentations. Dans ce manuscrit, nous abordons deux aspects de cette problématique, centrés autour de l'attaque et de la défense.

Nous avons dévoilé plusieurs attaques par canaux auxiliaires microarchitecturaux sur des implémentations de protocoles PAKE. En particulier, nous avons exposé des attaques sur Dragonfly, utilisé dans la nouvelle norme Wi-Fi WPA3, et SRP, déployé dans de nombreux logiciel tels que ProtonMail ou Apple HomeKit.

Nous avons également exploré le manque d'utilisation par les développeur·euse·s d'outil permettant de détecter de telles attaques. Nous avons questionné des personnes impliqué·e·s dans différents projets cryptographiques afin d'identifier l'origine de ce manque. De leur réponses, nous avons émis des recommandations.

Enfin, dans l'optique de mettre fin à la spirale d'attaques-correction sur les implémentations de Dragonfly, nous avons fournis une implémentation formellement vérifiée de la couche cryptographique du protocole, dont l'exécution est indépendante des secrets.

Title: Cryptography in the Wild: The Security of Cryptographic Implementations

Keywords: PAKE, cache attack, Dragonfly, SRP

Abstract: Side-channel attacks are daunting for cryptographic implementations. Despite past attacks, and the proliferation of verification tools, these attacks still affect many implementations. In this manuscript, we address two aspects of this problem, centered around attack and defense.

We unveil several microarchitectural side-channel attacks on implementations of PAKE protocols. In particular, we exposed attacks on Dragonfly, used in the new Wi-Fi standard WPA3, and SRP, deployed in many software such as ProtonMail or Apple HomeKit.

We also explored the lack of use by developers of tools to detect such attacks. We questioned developers from various cryptographic projects to identify the origin of this lack. From their answers, we issued recommendations.

Finally, in order to stop the spiral of attack-patch on Dragonfly implementations, we provide a formally verified implementation of the cryptographic layer of the protocol, whose execution is secret-independent.