THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National des Sciences Appliquées de Toulouse (INSA Toulouse)

Discipline ou spécialité :

Réseau Télécommunications Système et Architecture

Présentée et soutenue par :
Alberdi Ion

le : vendredi 9 avril 2010

Titre :

Malicious Trafic Observation Using a Framework to Parallelize and Compose Midpoint Inspection Devices (Observation du trafic malveillant en utilisant un cadriciel permettant la composition d'inspecteurs de point d'interconnexion).

JURY

Benzekri Abdelmalek, Blanc-Talon Jacques, Dacier Marc, Mellouk Abdelhamid, Mé Ludovic

---

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
OLC/TSF LAAS-CNRS

Directeur(s) de Thèse :

Nicomette Vincent, Owezarski Philippe

Rapporteurs :

Debar Hervé, Festor Olivier

Acknowledgments I would like to first thank my Ph.D advisors for trusting me enough to let me go in the direction I wanted, in spite of the craziness involved. Moreover, once the first prototype was running, all the debates concerning the definition and global purpose of Luth, were worth noticing. I could not have presented this Ph.D without these discussions, which I now qualify as fundamentals.

Among these debates, the first one has of course a special place in my memory. Indeed, I'd like to thank Béatrice Barbe for her great work, comments, advices and for all the pleasure I had working with her during her 3 month M1 internship.

A quite nostalgic thank goes to those who I consider to be my 4 most influential science teachers. They have been the ones that taught me, both the most important and subjectively the most difficult quality of a proper scientist: rigor. For this reason, I would like to thank my last high-school mathematics teacher in Bernat Etxepare, Myriam Junca. She was the first one to identify and to try to correct my lack of rigor. Following her advices, I had then the chance of learning how to demonstrate a mathematical fact with my two teachers of MPSI and MP, highly selective classes to prepare for the competitive exams to the "Grandes Ecoles" at Louis-Barthou. There, I assisted in the lectures of Olivier Ginoux, and Jean Pierre Lahargue, where they taught me ideas like: "intuition is good, but it's not enough". Finally, I would like to thank a colleague I met in MPSI, who happened to become a close friend. This classmate has been the one who taught me computer science could be rigorous too[1]. That's why I'd like to thank Sebastien Mondet, for having introduced OCaml to me, for having reviewed the most difficult chapter of my Ph.D, and for all the discussions we had by e-mail, from quite a lot of different locations in the world, about how we could make Internet software less catastrophic. This Ph.D thesis owes you a lot, thanks Sebastien.

Last but not least, I would like to thank my family and friends in general, for all the support they offered during this Ph.D. Without the energy they gave me, I could not have overcome all the challenges I had to face.

[1] Which is, from my point of view, not as obvious as it can seem, at least after my first 4 years of studying computer science from the Internet or from classes in engineering schools and universities.

List of acronyms

- **ACL** : Access Control List.
- **ADPS** : Automatic Data Processing System.
- **ARPA** : Advanced Research Project Agency.
- **ARPANET** : ARPA network.
- **AV** : Anti Virus.
- **C&C** : Command and Control.
- **CIDR** : Classless Inter-Domain Routing.
- **CPU** : Central Processing Unit.
- **DEP** : Data Execution Prevention.
- **DOS** : Denial of Service.
- **DDOS** : Distributed DOS.
- **DNS** : Domain Name System.
- **EBNF** : Extended Backus-Naur Form.
- **FTP** : File Transfer Protocol.
- **GC** : Garbage Collector.
- **GADT** : Generalized Algebraic Data Types.
- **HTTP** : Hypertext Transfert Protocol.
- **IDS** : Intrusion Detection System.
- **IPS** : Intrusion Prevention System.
- **IP** : Internet Protocol.
- **IRC** : Internet Relay Chat.
- **ISP** : Internet Service Provider.
- **I/O** : Input/Output.
- **KVM** : Kernel-based Virtual Machine.

- **LAN** : Local Area Network.
- **LOC** : Line(s) of Code.
- **MTU** : Maximum Transmission unit.
- **MI** : Midpoint Inspector.
- **NAT** : Network Address Translator.
- **NIC** : Network Interface Card.
- **NIDS/NIPS** : Network IDS/IPS.
- **OSI** : Open Systems Interconnection.
- **PC** : Program Counter.
- **PS** : Protection System.
- **RDBMS** : Relational Database Management System.
- **RFC** : Request For Comments.
- **SCTP** : Stream Control Transmission Protocol.
- **SMTP** : Simple Mail Transfer Protocol.
- **SQL** : Structured Query Language.
- **STN** : Switched Telephone Network.
- **TCB** : Transmission Control Block.
- **TCP** : Transmission Control Protocol.
- **UTM** : Universal Turing Machine.
- **VM** : Virtual Machine.
- **VoIP** : Voice Over IP.
- **WEP** : Wired Equivalent Privacy.

Contents

Introduction

> "The 3/4 of scientific progress is aimed at neutralizing the side-effects induced by the last quarter." Claude Levi-Strauss.

The idea that a network of computers could be used to augment human thinking came from J.C.R. Licklider, a psychologist and computer scientist when he was working at the *Advanced Research Projects Agency* (ARPA) (1962-1964). Mr Licklider's wanted to ease the communications among ARPA research contractors to make their research more efficient[2]. From this idea came the *ARPA network* (ARPANET) and finally the Internet network. Started as a collaboration tool among predefined users, the Internet has become a network connecting unpredictable users with hardly predictable purposes. Indeed a user can nowadays access the Internet from multiple terminals[3], interconnected through various mediums[4] available from a wide range of locations[5]. Once connected, thanks to Web sites and related applications, file sharing networks, text and live chatting programs, *Voice Over IP* (VoIP) networks, or video streaming sites, the Internet gives its users the opportunity to access and publish a hardly manageable amount and type of information. From the network to improve the knowledge of ARPA society, Internet is aimed at becoming, if not already, the network to build information societies at large. Among others, using the Internet, a user can read or publish the recipe of a tiramisu as early edition of digital newspapers, download the last blockbuster from Hollywood or exchange child pornography. These first types of behaviours are the result of an extensive use of the possibilities offered by a functioning Internet. The juridical, scientific and technical efforts to limit some of such behaviours issue polemical debate on topics such as copyright laws [52], freedom of speech [102], or anonymity and privacy rights on the Internet [194]. These problems are however not the only ones to be addressed. Indeed faults rise in the Internet too. Figure 1 plots the percentage of the time a service offered by one of the four telecommunication network kinds was functional during a year. It exhibits the very low availability of the services offered by the Internet network [127].

Let's try to explain the causes of such poor results. The growth of the Internet has been driven by a constant desire for new functionalities. Indeed the Internet customers are much more willing to quickly get a new buggy service, rather than having to wait for a version with less bugs. Therefore, the development of the Internet has been driven by a race for innovation: new services should be developed as fast as possible. If we take a look back at Levi-Strauss's quote, the more functionalities you have, the more undesirable side-effects you'll probably get. Even more, if the new functionalities creation rate is high. Indeed, after having implemented the easiest services, innovating means, most of the time, creating more complex services. Therefore, this race makes developers create more complex systems while enduring the still pressuring time to market. Understanding why *Switched Telephone Networks* (STN) availability outperforms Internet's is therefore straightforward. The faults, i.e. the non wanted behaviours of the Internet services, should be classified depending whether they are due to accidental or malicious activities. This thesis does not address the resolution of unintentional incidents. The following section will describe the correlation between bugs in software and malicious activities.

[2] Internet Pioneers `http://www.ibiblio.org/pioneers/`

[3] Computers, laptops, pdas, cellphones...

[4] Whether with wired or wireless technologies.

[5] Homes, schools, labs, streets, restaurants, trains, planes...

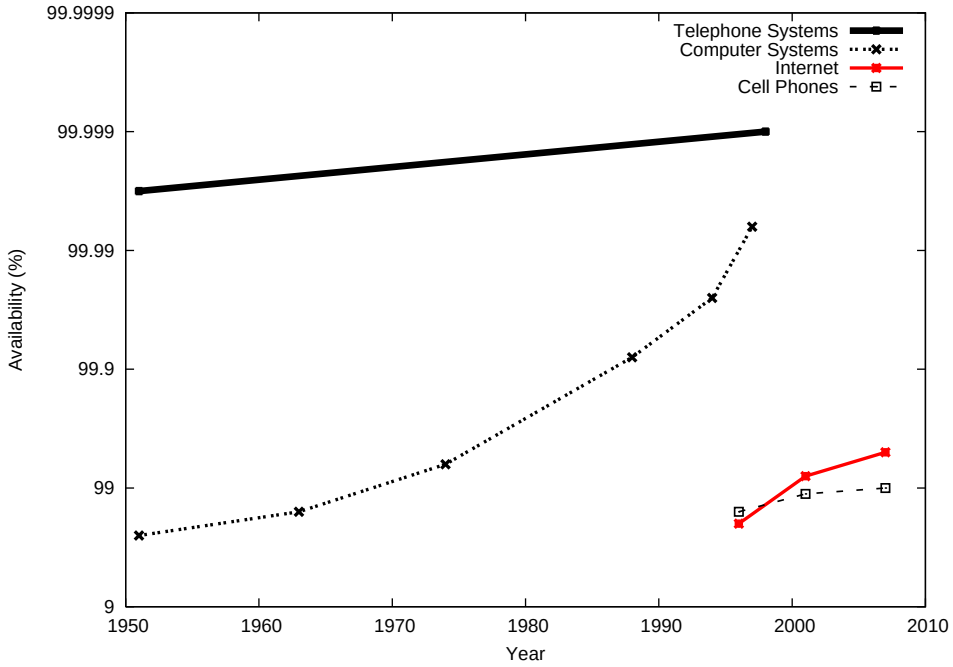


Figure 1: Evolution of the availability of networks

0.1 How The Race for Innovation Opens Doors to Malicious Activities in the Internet

In 1983, in an article called "Beware: Hackers at play", Newsweek published one of the earliest article mentioning the hacker term [3]. Starting from this article to the last episode of "Die Hard tetra-logy", numerous mass-media products popularised the idea that some computer wizards, whether acting for the greater good (Matrix, Independence Day), for challenge (Track Down) or for money (Golden Eye), could break into **any** computer interconnected to the Internet. Before going further, the reader should be informed that the definition of the hacker term is polemical [210]. To avoid possible misunderstandings someone who tries to break into a computer will be referred as an *intruder*. Similarly, people conducting malicious activities in the Internet at large will be referred as *attacker*s. For a non negligible part of Internet users, a sufficiently skilled intruder could break into any interconnected computer. When assessing this latter fact, the responsibility of a successful intrusion is subjectively given to the expertise of the intruder rather than to the errors done by computer-science users, developers, administrators, designers and researchers. Therefore, we do think that the **first** step to make **before** thinking of a solution to the problem, is to consider another paradigm. That's why, this thesis **strongly** argues against this latter **fatalist** and quite **comfortable** idea. Indeed, even the most skilled intruder needs a breach in the targeted computer systems to achieve his goals. Without a breach he **will not** break into it. A correctly designed and implemented system fulfils its specifications. Therefore, unless the presence of breaches

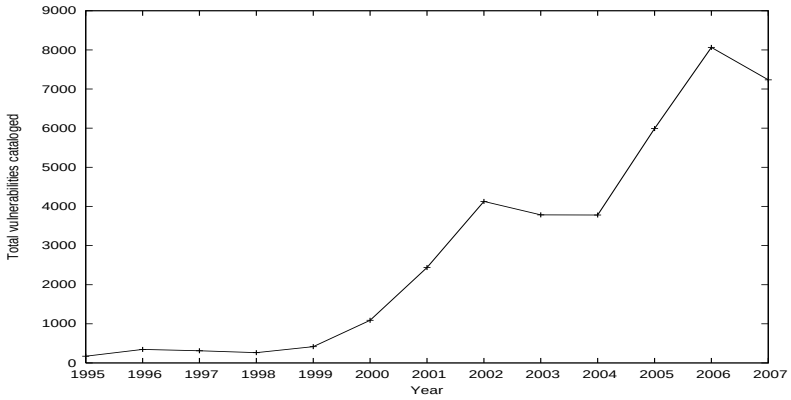


Figure 2: Evolution of number of vulnerabilities reported by the US-CERT

is specified[6], a correct software does not by definition have **any** breach, making intrusions **impossible**. However, software is developed in function of the race for innovation, and therefore has bugs. Even if not all computer bugs open exploitable breaches, some special bugs, called vulnerabilities, do. Computer science is still a young discipline, and formal verification of computer software and networks have probably still a long path to follow[7]. Therefore, it is not surprising that the vast majority of systems connected to the Internet, still focused on the race for innovation, has bugs and more precisely vulnerabilities as shown in Figure 2. The apparition and evolution of malicious activity in the Internet is therefore a logic side-effect of the race for innovation, carrying **more** responsibility than intruders' expertise. We have in this section assessed that intrusions or attacks are possible, but not the order of magnitude at which they can influence the functioning of the Internet. The following section will explain why even a single malicious Internet user can have a non negligible impact.

0.2 Von Neumann Machines and the End to End Principle: How to Automate Malicious Activities in the Internet

One of the main characteristics of a computing device is its computational capability, i.e. what it is capable to compute. For example cash machines used in most grocery stores can compute arithmetic operations like additions, subtractions or multiplications, but cannot compute integrals like $\int_{\sqrt{3}}^{e^5} x^2 dx$. Such integrals should therefore be computed by hand or by a new machine, providing it has already been designed and built. The design of computing devices reached a computing capability modeled by *Universal Turing Machines* (UTM) [231] and implemented by von Neumann machines [236]. These machines represent the great majority of nowadays computers. As depicted in [228] these machines have computational limits that new theoretical[8] machines, *hyper-computers*, overcome. Even if limited, UTMs

[6] Is the exploit of a specified vulnerability an intrusion ? This issue is open to debate, indeed errors can appear in specifications too. However, the day computer scientists will "debug" specifications rather than implementations, we do think the problem of widespread intrusions will become negligible, if not forgotten.

[7] It should however be noticed that the length of this path is strongly correlated to the priorities given to computer-science research and developing.

[8] Some theories like the strong version of Curch-Turing thesis argue such machine cannot be physically implemented. The debate is however still open.

have the peculiar capability of computing other Turing machines [183]. In other words, a computer program running on a von Neumann machine, can compute and execute another computer program.

The interconnection of computers has been driven by the "End to End principle" [216]. Whenever possible, communications protocol operations should be defined to occur at the end-points of a communications system, or as close as possible to the resource being controlled. Therefore among others, this design principle implies that an endpoint in the Internet can talk to any other interconnected endpoint.

These two facts make the following scenario possible. After identifying an exploitable breach on an endpoint system, a computer scientist can develop a computer software that automatically exploits this breach for its profit. Then, the malicious program could generate a new version of itself using the hijacked von Neumann machine, ask the attacked endpoint located in the Internet to download and execute this new program, which will try to attack a new target, etc.

Robert T.Morris, a first year computer science graduate student at Cornell University (USA), launched in 1988 what is known to be the first malicious and significant Internet worm [117]. A worm is a self-replicating computer program that sends itself to other interconnected computer without any user intervention. The Morris worm infected about 6000 interconnected computers. This idea was interesting enough to inspire a non negligible part of computer scientists as described in [229]. After having explained why even a single user can have a non negligible impact on the Internet, we will describe in the following section the rationale behind the solution we have designed in this Ph.D.

0.3 Concerns on the Internet at the Begining of the XXI^{st} Century

0.3.1 Botnets: Automating and Coordinating Compromised End Hosts' Actions

Nowadays, one of the most visible Internet malicious activity is driven by a slight modification of such network worms. *Botnets* are network of *bots*, when a *bot* is an end-host running a *botware*, a computer software used to remotely control the *bot*. A master, located somewhere in the Internet, sends to its bots orders through a communication protocol named Command and Control (C&C). The functionalities offered by such bots can be vast, as described in Figure 3. They can snif the network packets (to gather confidential information like login/password couples), try to propagate by perpetrating intrusions on other computers, send spam, or launch network attacks to flood a distant server by performing a *Distributed Denial Of Service* (DDOS). According to some estimations, performing such activities can generate profits that reach hundreds of millions of dollars [181, 149].

Financial gains does not seem to be the only motivation behind botnet maintenance. The correlation between geopolitical actions and distributed denial of services that occurred in Georgia [50] and Estonia [226], together with the discovery of an International cyber-spy network targeting high value political and economical computers based almost entirely in China [51] seems to indicate that some botnets are used as a political tool. This concern has been taken into account by among others, Barack Obama's administration which has decided to enforce some American cyber-infrastructures [131].

0.3.2 Monitoring End-Host Software From Midpoint Devices

The side effects due to distributed and coordinated malicious activity in the Internet creates a threat for the future of the Internet. As we said earlier, the apparition of such activity has two main causes:

- The race for innovation that develops and deploy incorrect software executed on von Neumann machines.

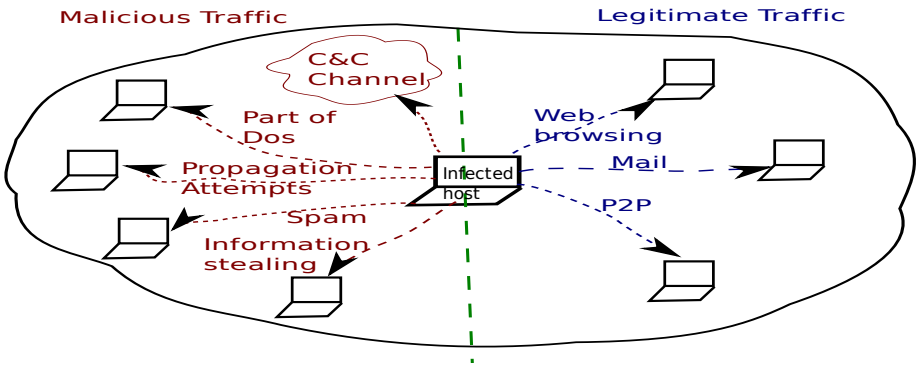


Figure 3: Network communications used by a bot.

- The End-To-End principle that delegates the functioning of the Internet to its end-host which are nowadays, driven by incorrect software.

In chapter 1 after presenting some ways to perpetrate intrusion and attacks, we present a survey of the solutions against botnet proliferation. First we present the existing methodologies to develop correct software. Then, considering that such solutions do not seem to be adopted yet, we present the different methods to monitor suspicious computer programs. Finally among these software, we will present a recent kind of software that aims at observing malicious activity. As astonishing as it can be, the software to monitor or observe suspicious computer programs has, up to our knowledge, shown little interest to the technologies to develop correct software. This lack of interest creates some quite logically distorted situations where the software to defend a computer, for example an antivirus, opens a breach in the computer it is supposed to defend [141]. In this Ph.D. we state that Internet's end-host software and systems cannot be trusted yet. Therefore, until correct software developing technologies are popular enough to be widely adopted, end-host software must be monitored. Taking into consideration that a monitoring software in corrupted hosts could be modified by the intruder gaining the control of the computer, we affirm that such monitoring should be performed by a correct software or device, put in end-hosts' interconnection points, what we call midpoints. As depicted in chapter 2, the tool we design enables to perform a combination of all usual midpoint computations, ids/ips, firewall, protocol identifier, shaper, into a single software. To do so, instead of creating a monolithic design pattern, we split the actions to perform into Midpoint Inspectors, MIs, that can be seen as mini ips/ids/firewall, etc. A language designed during this Ph.D enables to compose and parallelize a set of interoperable MI-s, to produce a unique midpoint inspection algorithm, based on an inspection tree. To check the validity of the computations, we provide algorithms and their proofs to verify the correction, optimality and to deduce the missing parts of a given inspection tree. We implement this idea on a prototype developed in a type-safe language OCaml, and benchmark its performances. The latter appear to be sufficient for the two reported experiments, the monitoring of:

- malware downloading honeypots, and
- dynamically analysed malware's network communications.

Chapter 3 deals with malware downloading. Malware downloading honeypots are end-host software emulating the behaviour of the victim of a computer intrusion, hoping that the intrusion aims to upload

and execute a malware into the targeted system. If so, such honeypot emulates the download and stores the malware without executing it. Considering that trusting end-host software is what conducted to the apparition of botnets, we first check whether malware downloading honeypots can be trusted. After finding ways to use malware downloading honeypots as computer intrusion proxies, we configure our prototype to reject these malicious actions while accepting behaviours that are 'normally' considered as malicious and therefore rejected. Chapter 4 describes the platform designed to perform , as safely as wanted, the dynamic analysis of a malware talking to the Internet. Taking into account that attacking third party's Internet end-hosts is illegal, letting a malware talk to the Internet presents some risks. That's why up to now such analysis has been performed in an emulated Internet environment or in non detailed platforms where little attention is given to the potential side effects of the analysis. The tool designed in this Ph.D. proposes a solution to this problematic and opens a door to the benefits of such dynamic analysis. We detail in the last chapter the first two explored benefits:

- malware clustering,
- automatic intrusion detection signature creation.

1 Monitoring Computer Programs

> "I'll let you in on a secret: my pet hamster did all the coding. I was just a channel, a 'front' if you will, in my pet's grand plan. So, don't blame me if there are bugs. Blame the cute, furry one." Harald Welte.

To have a better understanding on malicious activities, we devote the first section of this chapter to the explanation of how most of intrusions and attacks work. In a second section we present the different existing approaches to limit malicious activities in the Internet. Finally, the last section summarises these approaches, to introduce the solution proposed in this Ph.D.

1.1 Understanding Vulnerabilities and Network Attacks

As we stated earlier, different breaches can be used by an intruder to take the control of another computer. Similarly, different ways can be followed to conduct network attacks in the Internet. Therefore, to understand some of the choices we made, we first present the functioning of the most widely reported intrusions and attacks.

1.1.1 How to Take the Control of an End-Host Software or System

From a juridical point of view, an intrusion is said to be "an illegal entry upon or appropriation of the property of another"[1]. By browsing the French penal code, the intrusion on an automatic data processing system (ADPS) is not very clear when dealing with accessible ADPS. Does a user connecting to an open wifi hot spot commit an intrusion punished by the 323-1[2] article of the French penal code? We do think that from a more pragmatic point of view, there is a single way to ensure that a user owning a wifi hotspot does not get unsolicited users: teach him how to use a solution to implement the policy he wants. Without such policy, a user has no way to know if a hotspot is deliberately shared or not. This kind of decisions therefore limits the usability of deliberately shared hotspots, and therefore the accessibility to the Internet. That's why we have chosen not to address the situations where an interconnected entity uses a non policed service.

1.1.1.1 Asking End-Users to Execute a Program

Someone does not need to be an expert to make a non accessible computer execute the instructions he wants. Indeed, asking one logged enduser to download and execute a software hosted somewhere in

[1] http://www.thefreedictionary.com/intrusion

[2] http://www.legifrance.gouv.fr/affichCode.do?idArticle=LEGIARTI000006418316&idSectionTA=LEGISCTA000006149839&cidTexte=LEGITEXT000006070719&dateTexte=20090922

the Internet is sometimes sufficient. Indeed, a web-site or mail could advertise the free download of a computer program with a malware hidden in the advertised binary. One of the best managed botnet that disappeared by the end of 2007 for unknown reasons used this kind of obvious propagation method [135].

1.1.1.2 Bypassing Authentication Schemes

Different breaches could be exploited to bypass authentication schemes. As shown by [57], dictionary attacks are a popular way to bypass some authentication schemes. Empirical studies show that in a login/password authentication scheme, a large fraction of the users choose passwords from a restricted lexical domain [200]. The breach consists then in trying all the couples in a subset of such domain. The design of the authentication protocol can present a flaw too, giving a malicious user the possibility of impersonating itself. The WEP, that aimed at giving Wired Equivalent Privacy to Wifi networks, is a famous example of such buggy protocol [223]. The implementation of authentication protocols can be buggy too. We can mention the flaw in the OpenSSL package provided by the Debian project as an example. This package generated predictable cryptographic keys[3], and therefore falsified all the properties ensured by the implementations of protocols relying on the non predictability of such keys. Finally the assumption behind the theory on which the authentication scheme is based on, could become false too.

1.1.1.3 Code Injection: The Undesirable Side Effects of von Neumann Machines and Languages

The SANS institute depicts in [142] what are the Top Cyber Security Risks computers face in the Internet. All these risks are different forms of what we will call *code injection attack*. In this section we will show that *code injection attacks* are strictly correlated to a popular paradigm invented by von Neumann Machines: the mixing of instructions and data. Before going further we will describe the reasons behind such a paradigm.

The main thing computer scientists have in common is their desire to automatise one process. Instead of making repetitive tasks by hand, computer scientists build automatisation systems. The 19-year-old Blaise Pascal was working with his father, computing the taxes some people needed to pay. He was so tired of computing all the additions by hand, that he developed a device to make this tedious task. The Pascaline was born in 1652. In 1821 Charles Babbage, decided a computing device should be built to fill the mathematical (logarithm, trig) tables used by navigators, surveyors, astronomers. Indeed, these tables were previously computed by "human-computers" leading to a lot of errors. Babbage had there the visionary idea of building a programmable computing device, even if the technology available at this time was not advanced enough to realise the machine [225, 86]. Charles Babbage's idea was materialised by Howard Aiken in 1944 with the Harvard Mark I [71]. This computer was able to automatise addition/-subtraction/multiplication and division computing operations. A computer program describing the list of computations to make was given in its input. Harvard Mark I then returned the result. However this machine was not able to compute computer programs which were still needed to be calculated by hand. A slightly different approach was developed by Alan Turing since 1936. The latter formalised the definition of a universal computing machine, the UTM in [231], which is considered to be the starting point of computer science. In 1945, John von Neumann designed the architecture needed to build such universal computing machine. The first von Neumann machine, the EDVAC was built in 1949 [236]. Most of the

[3] CVE-2008-0166

computer machines we use everyday are new versions of this machine. Modern computers are therefore the result of a joint effort to automatise computation tasks. As stated by John Backus in [63]: "In its simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store)." Another paragraph of this Turing Award[4] winning article written in 1978 is worth quoting: "When von Neumann and others conceived it thirty years ago, it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of computer with this thirty year old concept". Indeed, 60 years later, the notion of computer remains the same.

What makes von Neumann computers different from Harvard computers is that the latter has two different stores: one for storing instructions, the other one for storing data. For example, let's take the $1+2$ computation. Here, we have two data elements, 1 and 2, and one instruction, $+$. To compute such addition in a Harvard computer, the 1 and 2 values are stored in the data store at addresses @a and @b. The instruction

```
ADD @a @b @c
```

which adds what is in @a and @b and stores the result in @c, is entered in the program store. Then, the machine is executed and the user can finally read 3 in $@c$. To compute additions like $4+(1+2)$, a computer uses what is called a Program Counter (PC): a pointer to the next instruction to execute. That way, instead of a single instruction, computers can compute sequences of instructions.

Listing 1.1: Addition

```
1.WRITE 1 @a /* write the value 1 in data memory a */
2.WRITE 2 @b
3.ADD @a @b @c /* add what is in a and b, */
               /* and write the result */
               /* in data memory c */
4.WRITE @c @b
5.WRITE 4 @a
6.ADD @a @b @c
```

To do so, the Program Counter, PC, is initialised at 1: the address of the first instruction. The program then computes it, and increments PC to execute the instruction at address 2, and so on. However, let's say the current program is a software to train a user in making correct additions. Let's say this software asks the user the result of $4+(1+2)$ and prints *OK* if the user inputs 7, and *NOK* else. To do this kind of branching, the program needs to set some values to the program counter PC.

Listing 1.2: Addition teacher

```
x.SUB @c @input @c. /* SUB is the analogue of ADD for subtractions */
y.WRITE @e @nxt.
z.IFCISZEROSETPC @nxt. /* Sets PC to the value written in data */
                       /* memory nxt if what is in data memory */
                       /* c is zero */
g.PRINTOK
h.EXIT
e.PRINTNOK
```

[4]Turing awards are the equivalent of the Nobel prizes for computer scientists.

```
f . EXIT .
```

After having illustrated these notions, we'd like to emphasize the following properties of Harvard machines:

- the CPU can load instructions only from program memory,
- the CPU can only write in data memory,
- the program counter, pointing to the next instruction to execute, can only access program memory.

Therefore, as the CPU cannot execute what is in the data memory, and as the only location the CPU can write at is the data memory, the Harvard computers CPU cannot write executable instructions. For these reasons, Harvard machines cannot *directly* execute a computed computer programs, as a program must be loaded by third party into the instruction memory.

Unlike Harvard machines, von Neumann computers mix the data and the instructions in the same memory. Depending on the value of the program counter, which evolves during the execution of program, the value read at some address will be interpreted as data or instruction by the machine. Mixing instructions and data therefore enables the CPU to compute and store instructions together with data in its memory, which is sufficient to compute executable computer programs. However, let's say our computer program is a bogus music player. With a particular entry, the music player's program counter will be set to the value at an address a1, used to store a part of the song to play. An intruder could then craft a special song, which:

- triggers this bug,
- sets PC to a value pointing to the middle of the song in the memory at the address a1,
- writes in a1 (which is part of the song he has specially crafted) instructions to download and execute a malware in the Internet,

Like that, when playing the specially crafted song, the music player will finally execute the instructions developed by the intruder. The so called stack buffer overflows [189], heap overflows [61, 148], integer overflows [188, 81], or format string vulnerabilities [123] exploit the bugs of programs together with the functionalities of von Neumann machines.

Nowadays, computer software is however not directly written using von Neumann machines instructions' set. Modern software is usually written in higher-level languages, that are then translated into a sequence of von Neumann machine instructions using a special computer software called *compiler*. These higher-level languages aim at easing the development of software, and therefore reduce the software developing time. Before going further, we'd like here to quote John Backus again, "Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor-the von Neumann computer, [...] and their lack of useful mathematical properties for reasoning about programs."

What should be emphasized by this still valid 40 years old citation is that the philosophy behind von Neumann machines is still popular. Von Neumann machine's creators, focused its design on its expressiveness rather that on possible ways to extract properties about its possible behaviours. The mixing of data and code was just a single side effect of the global wish to extend the boundaries of what could be automatically computed by a real machine, without worrying about the possible bugs such

computations could have. Indeed, expecting two revolutions from the same people living a single life is not reasonable.

Nevertheless, nowadays computer scientists know what bugs are and their consequences. Therefore it is hardly believable that the ideas in this 40 years old article have still so little impact in the vast majority of computer-science courses, and therefore on computer scientists minds. Indeed, C [5], C++ [6], Java [16], PHP [30], Python [31], Javascript [17], Ruby [41] i.e. nowadays most popular computer languages, are von Neumann inspired languages. The side effects of this trend are however more logic. To illustrate the impact a language can have on our reasoning capability, we should remind that there are quite a lot of ways to express the numbers we use everyday. Let us make the mental exercise of living in a society were basic arithmetic are taught using Roman numerals. One teacher will among others, have to check the correction of additions like $MDCCIIIII + X = DDMCIV$. First, even if such numbers can be written, the teacher should consider the numbers $MDCCIIIII$ and $DDMCIV$ as invalid, and rewrite them following Roman numerals rules as $MDCCV$ and CIV. Then, to verify $MDCCV + X$, the teacher should make some kind of mental exercise to emulate this addition, and compute the result, without ways to guaranteeing its correction. Logically, students will make quite a lot of errors in their additions, and teachers will make errors in their correction of evaluations too. If we translate the latter exercise into Arabic numerals, it's easier to verify that $1525 + 10 = 104$ is incorrect. But our point of view is subjective as we all have learnt to compute arithmetic using Arabic numerals. Whatever subjective it can be, we do make a very close parallel between the article of John Backus, criticising von Neumann style languages and advertising for other paradigms[5] and what a Roman arithmetician would have written about Roman numerals after having discovered Arabic numerals, i.e. what Fibonacci actually wrote in "Liber Abaci" in the early of 13th century [219]. Von Neumann inspired languages are therefore error prone. Moreover if such languages do not neither differentiate code and data, higher level code injections will be possible too.

As described in [224], nowadays web applications, i.e the software driving web-sites behaviour, use an imbrication of different von Neumann languages to produce dynamically computed web pages. Some Web applications use databases to store data like logins and passwords for authentication purposes. In this case, a first language (most of the time SQL), is used to implement the operations on the database. Another language is used to handle the requests sent by the user's web browser and produce dynamic web-pages. A lot of different technologies implement similar functionalities, therefore we abstract all of them as WebL. The following SQL statement enables to select all the users in the database containing the login $u1$ and password $p1$:

```
SELECT * FROM accounts WHERE name ='u1' AND password='p1'
```

The Web application in itself extracts the user and password from the request it just received. Then, it computes a similar SQL statement to check if the login and password couple sent exists in the database. Indeed, if these requests are not in the database, the couple is invalid and the user failed to identify as a valid user. In WebL, the program to do so will have this kind of pattern:

Listing 1.3: Web application code

```
request=getRequest();
user=extractUserFromRequest(request);
passwd=extractPasswdFromRequest(request);
sqlQuery=buildSelectRequest(u1=value(user),p1=value(passwd));
```

[5] In his case, functional languages.

```
res=executeSqlRequest(sqlQuery);
if isEmpty(res) {
   disconnectUser(); displayFailurePage(); }
else {
     connectUser(); displayHomePage(); }
```

By making an analogy with our previous example, we can see that the

```
executeSqlRequest
```

function as a von Neumann machine, where the computed results are function of data (u1,p1) and instructions

```
(SELECT , FROM , WHERE , AND)
```

mixed in the global memory sqlQuery. The same way von Neumann machines do not differentiate data from instructions, the *webL* von Neumann inspired language, does not check whether the storage user contains instructions. Therefore, the same way a malicious user can craft special songs, a malicious user could use the

```
(anybody,' OR 1=1)
```

couple to authenticate himself. WebL will then compute the request:

```
SELECT * FROM accounts WHERE name='anybody' AND password='' OR 1=1
```

This request returns all the accounts of the database, i.e a non empty result, and therefore the malicious user is logged in. Similar other injection techniques can be applied to conduct this kind of intrusions, as web applications mixes a lot of different languages. Indeed, Cross-Site scripting [153], XPath injection or Shell injections [224] are other examples of such behaviour.

1.1.1.4 Summary

We have in this subsection presented some ways to conduct intrusion attacks. We have given a special attention to von Neumann machines functioning and ideas, to understand why bugs are omnipresent, and how the most popular kind of intrusion methods, i.e. code injection attacks were possible. In the following section we explain how, once taken the control of one or multiple Internet end-hosts, Internet services' availability can be attacked.

1.1.2 How One or Multiple End-Hosts can Attack Internet Services Availability

The other type of attack computer defence systems need to deal with are the ones targeting the functioning of Internet services: *Denial of Services* (DOS). A very effective denial of service consists in triggering a bug leading to the crash of one application. For example, if a bug sets the Program Counter of a server program to an invalid value, the program will crash and stop answering user requests. The same way, bugs in protocol design or implementation could cause the misbehaviour of some nodes leading to a denial of service too. These techniques are however application or protocol dependant [157, 92, 93]. That's why other more generic ways are popular. As stated in Section 0.2 the building of the Internet has been driven by the "End to End principle" [216]. Whenever possible, communications protocol operations should be defined to occur at the end-points of a communications system, or as close as possible to the resource being controlled. Indeed, two communicating endpoints, A and B, need to face a lot of problems when for example trying to reliably transfer a file. If the software running in A assumes that

the network will reliably transfer each packet to B, the software written in A will not work in a network dropping some packets. Therefore, the software designer has the reasonable temptation to take possible packet loss into account. For these reasons, the End to End principle says that designing a smart core network is unnecessarily costful as most probably redundant. Moreover, designing and implementing a perfect network could be useful for file transfer, but not for other use cases. For example, multimedia data transfer is much more concerned about the overall transmission delay than the packet loss rate. That's why Internet's core network just focuses on transferring all data packets at its best. The endpoint, and most of the time the software running on end-host computers is responsible for managing all the possible errors. A direct side-effect of this design principle, is that an endpoint in the Internet can talk to any other interconnected endpoint. In the following three parts, we describe how the three pillars on which the great majority of Internet services are built upon can be attacked to reduce their availability.

1.1.2.1 Network Bandwidth

The Internet interconnects networks by means of different technologies. The Internet routers are responsible for transferring network data from one network to the other. However, these routers have a limited bandwidth. Therefore, responsible endpoints need to compute an estimation of the available bandwidth along the path to the other communication extremity. This kind of computation is performed by congestion control algorithms. However, a malicious user could choose to send garbage data, at a superior rate than the available bandwidth. This way, the router does not forward the data of other users to their destination, provoking their denial of service. This kind of attack has moreover collateral damages. Indeed, even if it aims at putting a single server down, all the computers interconnected by the flooded router will be out of service.

1.1.2.2 TCP's Pending Queues

Most of the services provided by the Internet use a particular protocol to reliably transfer data. The Transmission Control Protocol, TCP [35] is therefore a target of choice for conducting denial of service attacks. The same way a house collapses when its pillars are broken, putting the TCP stack of one application out of service, renders the service unavailable. The functioning of TCP is based on a client-server model: the server waits for client requests' apparition, and after some computations sends its result back. To connect to a TCP server, a TCP client must perform what is called a three-way handshake [35]. The clients sends a hello message parameterized by two identifiers: a first one to identify himself among other clients, the other one to identify each message sent during the TCP session. The TCP server answers back a hello message with the similar two identifiers and a third one to acknowledge the reception of the first message. After receiving this message, the client sends a last message to acknowledge the message from the server, finalising the handshake. Once this process has been achieved, the TCP session is said to be established, and the transmission of data can start. To implement such a communication behaviour, the TCP server software needs to keep in memory at least two queues:

Transmission Control Block (TCB) queue The queue containing the two identifiers used to differentiate TCP clients willing to establish a session.

Established session queue The queue to store all established clients. Indeed, much more information is needed in this queue, like all the bytes to be sent or sent but unacknowledged by the clients.

The storage memory of one server is limited, and therefore prone to denial of service attacks. One of the most popular denial of service attack, the TCP SYN Flood [94], tries to deplete the first queue. One

could argue that one client willing to conduct such attack is limited by the number of valid identifiers he could generate. Two arguments balance this remark:

- This limitation could be overcome by distributing the attack along the necessary number of nodes.
- The design of the Internet does not force one computer to use its valid identifiers. Therefore, a node could impersonate or *spoof* other nodes. The only side effect of spoofing is that the answer does not go back to the malicious client but to the spoofed address. However, to conduct a TCP Syn Flooding, the answer from the server is not necessary and the number of identifiers a malicious user could use is vast. Moreover, this spoofing technique makes it more difficult to identify which are the real computers behind the attacks.

Taking into account the resources needed to maintain ESTABLISHED sessions, applications using the TCP transport protocol limits the number of simultaneous active ones. The naptha attack [91] establishes the maximum number of simultaneous sessions an application could support and then sends data to make these sessions persist as long as possible. The figure 1.1 plots the number of packets and bytes going and coming back from a web-server during a simulation we conducted in our laboratory. The web-server answers the requests of 50 different clients. The number of ongoing packets and bytes are plotted every ten milliseconds. The figure 1.2 plots the same activity when a single client performs a naptha attack. During the attack all the legitimate clients could not connect to the server anymore, and no more web-pages are exchanged.

1.1.2.3 Computing Resources

Finally, computing resources could be targeted by asking one server to process excessive calculations. Indeed, some studies have predicted than in the evolution of dynamic web-server[6], their computing resources (CPU, memory, disk/database bandwidth, and I/O bandwidth) may become their main bottleneck [208]. Therefore, by flooding the server with requests demanding extensive use of such resources, a denial of service could be provoked too [150]. This technique is for example known to have been used by an extortion scheme [205] in 2004.

1.1.2.4 Summary

In this section we have presented how the correlation between intrusion possibility together with the implicit trust granted to Internet end-hosts creates a threat to Internet services availability. After having presented some of Internet's problems, we now present a panorama of proposed solutions.

1.2 Existing Solutions and their Limits

In this section we first explain a state of the art of the technologies to develop correct software. The first subsection aims, among others, at evaluating the feasibility of the first requirement of our study: developing correct software. Then, given the fact that little interest is given to such techniques, we present an enumeration of solutions to tolerate possible intrusions in software. After having noticed that such solutions are neither widely adopted, we describe two different software kinds: software to monitor computer programs and software to observe malicious activity.

[6] Servers that dynamically compute web-pages according to user requests

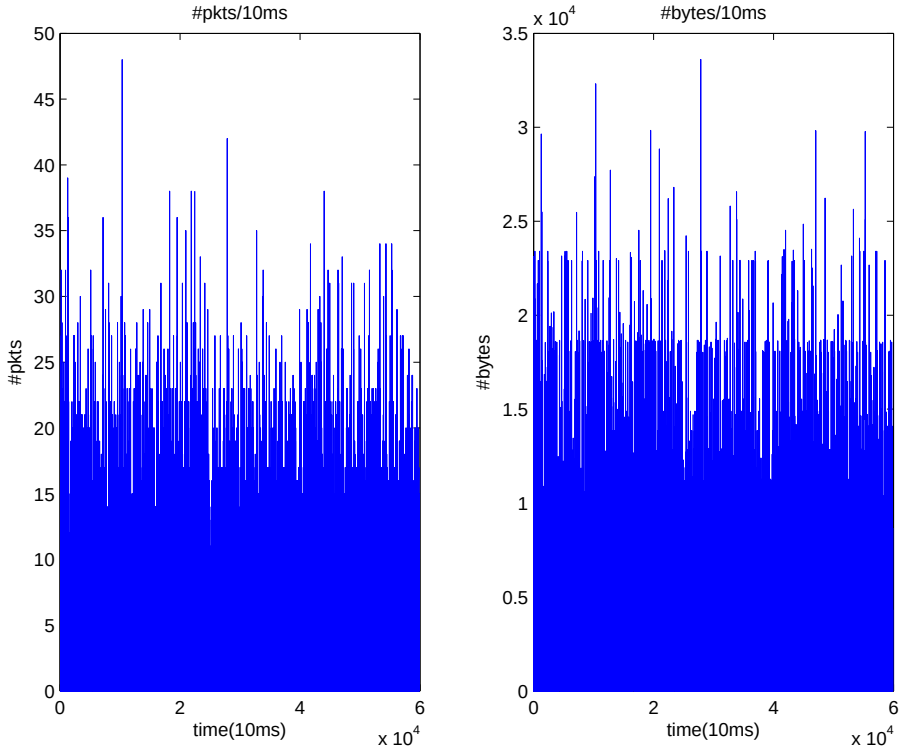


Figure 1.1: Number of packets and number of bytes time series during normal operations.

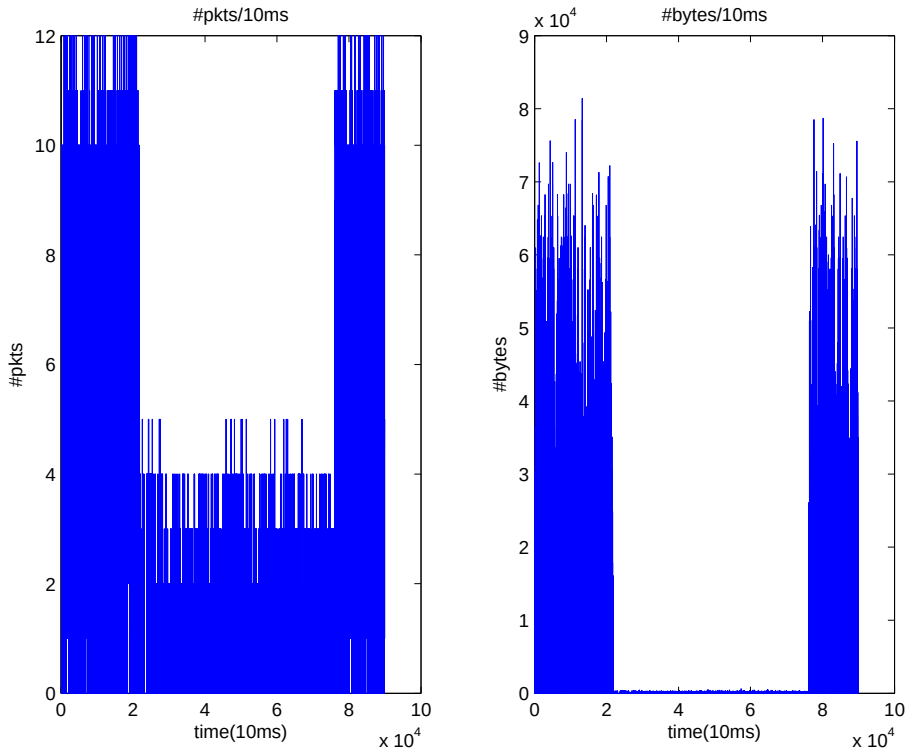


Figure 1.2: Number of packets and number of bytes time series during naptha attack. The abrupt drop in traffic volume shows the consequences of the successful denial of service.

1.2.1 Removing Exploitable Bugs: How to Develop Correct Software or at Least Code Injection Tolerant Systems

When defining the basic concepts and terminology of dependability, Laprie et al. defined in 1992 [161]: "A system failure occurs when the delivered service deviates from fulfilling the system function, the latter being what the system is intended for. An error is part of the system state which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a fault". More pragmatically, when describing how to build fault tolerant operating systems Peter J. Denning defined in 1976 [111]:

- a failure as an event at which a system violates its specifications,
- an error, as an item of information which, when processed by the normal algorithms of the systems, will produce a failure,
- a fault as a mechanical or algorithm defect which may generate an error.

In our specific case, critical vulnerabilities are therefore faults according to both definitions. Denning then defined:

- a correct system as a system free of faults and its internal data contain no error,
- a reliable system one were failures do not seriously impair its satisfactory operation.

The most elegant solution to the vulnerability exploitation problem, is therefore creating correct systems. Let us describe the different available solutions to create bug-less software.

A first approach consists in creating some computer languages presenting useful properties to check for the existence of bugs. Then a compiler, a particular computer software, will traduce this language into code understandable by the von Neumann machine, which can, as said earlier, be error prone. The purpose of these higher level language is more or less to restrict the set of the programs that can be expressed by von Neumann machines, to a subset presenting validated properties. One of the main tools used by such programming languages compilers are type systems. In [199] Pierce defines: "A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute". During our first mathematical classes, our teachers used to repeat that adding 2 potatoes and 3 tomatoes had no mathematical sense. We can only add 2 fruits and 3 other fruits, to have 5 fruits. Type-systems generalize this idea to computer programs. We should differentiate static type systems from dynamic ones. Static type systems check for properties before producing, and therefore executing the code. The Hindley-Milner algorithm proposes a very elegant solution to static type checking [175]. Moreover, it illustrates what could be accomplished following the idea behind John Backus, i.e. providing a software is written in a language designed to ease its correction verification, correction properties can then be automatically proved by a compiler. The algorithm of Hindley-Milner is based on a new computer language, ML, used to express, like other languages, the types and the computations to perform using variables of previously defined types. However unlike other languages and thanks to the latter algorithm, there is no need to explicitly mention the types of the variables. Indeed, the Hindley-Milner algorithm guesses the types of the variables, what is called type-inferring, **and** checks the correction of the guessed type system. For example, let us take the example of the following program written in the OCaml programming language, a successor of ML.

```
Printf.printf "%f" (2+3/2)
```

This program aims at printing the float result of the addition (2+3/2). When processed by the compiler, the implementation of the Hindley-Millner algorithm raises the following type error "This expression has type int but is here used with type float". Indeed, 2+3/2 is an integer computation, and its result is 3[7], not the float 3.5 as expected by the format "%f". The correct version of the program:

```
Printf.printf "%f" (2.0 +. 3.0/.2.0);;
```

correctly outputs 3.500000. However, the expressiveness of languages based on static type-system are limited. Dynamic type systems allow greater expressiveness by adding run-time checks. The object-oriented Eiffel language has for example been designed with this model in mind [173]. However, the guarantees offered by these systems are weaker, and offer partial correctness proof. Let's take the example of a dynamically checked computer software running the behaviour of an aeroplane. If a critical and unknown bug is dynamically discovered during a flight, the passengers about to crash will have little interest knowing the cause of the accident. Concerning the previously mentioned type-inferring, one could say that languages like Python do it too. Indeed, there is no need to explicitly mention the type of a variable in Python. However, Python uses what is called duck typing. Duck typing is a dynamic typing system that uses a variable's current set of properties to make type guesses. Therefore, Python uses the possible buggy semantic of the program to guess variables types, which can therefore be erroneous too. Let us write the previous example in Python:

```
print "%f" % (2+3/2,)
```

Instead of outputting an error, the Python interpreter prints 3.000000 considering that as a float was expected, the result of 2+3/2, 3, should be converted into a float. Programmers that have experienced some hours or days debugging such unexpected behaviours by browsing thousands of line of codes, will quite easily understand why the a priori simple idea behind the Hindley-Milner algorithm greatly eases the task of developing correct software. For the developers that need to go beyond the boundaries imposed by static type-system expressiveness, hybrid type checking proposes a synthesis of both of the approaches [120]. They perform static analysis when possible, and put dynamic checking otherwise.

Another more specific problem a computer software needs to deal with is memory management. Before trying to write somewhere in memory, the program should first check if there is available room, allocate the available region to prevent others to write there, and then liberate it in case others need it. This tedious task is the cause of the many computer bugs, like the causes of stack or heap overflows. That's why computer language designers developed garbage collectors. Garbage collectors automatize memory tasks and ensure that all memory accesses are guaranteed to be safe. This management has however a cost, as some computation is needed to prevent memory consumption leaks [62]. These great advances done in the computer language field however only apply to people willing to use them. Moreover, they don't apply to previously (or not yet) developed software in languages without such features. Therefore, other techniques have been used to verify properties over computer software in general.

The model checking aims at browsing all the possible states an abstraction of a computing system (usually computed by hand) could reach and check some properties over all of them [104, 105, 106]. This technique is limited by the combinatorial explosion of the number of states a system can reach. Indeed, this number reaches quite easily astronomical values, requiring an unreasonable time for a modern computer to explore. Some very interesting results have been obtained without overcoming the latter limitation in real time software and system engineering.

[7] Or 4 whether 3/2 is approximated to 1 or 2.

Static program analysis aims at analyzing a computer program without executing it[8]. In 1936, Alan Turing predicted that his universal machine could not resolve a particular problem. Indeed, his machine cannot say whether a program will halt with a given input [231, 184]. In other words, a computer program could not say by static analysis whether any other computer program will stop or indefinitely loop. Together with other problems, the latter proved that static analysis is undecidable [160]. However, abstract interpretation tackles the verification problem differently. As depicted by Cousot in [108], this field of research acknowledges that "since program verification is undecidable, computer aided program verification methods are always partial or incomplete. The undecidability or complexity is always solved by using some form of approximation. The purpose of abstract interpretation is to formalize this notion of approximation in a unified framework." Practical realisations using such technique have successfully checked the absence of out-of-bound array indexes, like buffer overflows, and logical correctness of integer and floating-point arithmetic operations, involved for example in integer overflows [80]. However, this technique still needs human expertise to obtain valuable properties on specific programs. Indeed, after verifying a specific type of program of 30.000 LOC (lines of code), the designed technique did not work on larger programs (250.000 LOC). To sum up, even if very promising, these technologies have still some path to follow before being applied in the design of a significant amount of Internet software.

The pioneer article written by Hoare in 1969 opened the path to formal verification of programs [133]. As depicted in a retrospective of his work [132] written in 2009, C.A.R Hoare says that: "My [...] job (1960 - 1968) was [...] to lead a team that implemented an early compiler for ALGOL 60. Our compiler was directly structured on the syntax of the language, so elegantly and so rigourously formalized as a context-free language. But the semantics of the language was even more important, and that was left informal in the language definition. It occurred to me that an elegant formalization might consist of a collection of axioms [...] that would be strong enough to enable programmers to discharge their responsibility to write correct and efficient programs." The idea behind this sentence is that, writing correct and efficient programs is a difficult enough responsibility so that a new technology needs to be designed to discharge programmers from doing it. The developing of programs to automate logical and mathematic proofs followed this path and permits nowadays, even if very expensive, the realizations of correct software. Two recent experiments are worth quoting. Leroy et al. conducted the development of a verified compiler using such a method [163]. The source language targeted by the project was a subset of C, called CLight. The developed compiler traduced programs written in CLight, to the language used by PPC computers, an implementation of a von Neumann machine. They used the Coq proof assistant[9] to certify the compiler. This tedious task was achieved by the equivalent of 2 person years (py) work to produce a project consisting of 13% of program code, and 87% of certification code. In [154], a team of about 13 people, implemented and formally proved another significant computer program: the micro-kernel of an operating system seL4. This other tedious task, consumed 2.2 py for developing and 20 py for designing the proof framework and writing the proofs of the program by means of the Isabelle proof assistant [10]. Thanks to the knowledge acquired during the experiment, the authors claim that a similar new realization (implementation + proof) could be reduced to 8 py. We do think that both results are worth emphasizing: the previous two **complex** programs are **correct**, which means among others, attacking them is **impossible**. This approach has however still a non negligible cost to be widely adopted in software programming industry. Nevertheless from our point of view, this approach

[8] Static type-systems are a particular case of static program analysis

[9] `http://coq.inria.fr/`

[10] `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`

is the **most promising** solution **ever** proposed against botnet proliferation, even if program correction checking has a wider purpose than computer security.

Correct systems are therefore still difficult to implement. To build reliable systems, one solutions is based on the system closure [111]: no action is permissible unless explicitly authorized. This principle inspires Protection Systems which aim, once authentication processes are done, at:

- blocking non authorized operations,
- limit the propagation of errors, in our cases, what an intruder could do.

Such mechanisms use three entities:

the subject an active entity executed from some user account,

the object a passive entity defined by a current state, and access functions,

rights the specification of the accesses a subject is allowed to perform on objects.

Therefore, a protection system checks the coherence between the rights and the accesses performed by a subject.

One way to limit the possibility of exploiting a vulnerability is to restrict the rights (read, write, execute), a subject or process on a von Neumann machine has on its memory (the object). Some operating system and computer architectures have therefore proposed functionalities, to put some order to the potential mess present in their storage. Memory segmentation enables to divide this global memory into different segments managed by access rights. That way, a segment can be configured to only store data, and therefore prevent a process from executing this content. This system can be seen as a compromise between von Neumann and Harvard architectures. Multics [151] or the IBM 801 operating systems [96] implemented such concepts about 40 years ago. The growing threat represented by computers intrusion made developers of some modern computer system willing to incorporate this idea. The Pax[11] and W^X [12] projects have been proposed to the Linux and OpenBSD operating systems. Intel and AMD processors, other implementations of von Neumann machines, implemented the NX/XD[13] technologies to facilitate such implementations too. The Windows family operating system used it to implement the Data Execution Prevention (DEP), since the Windows XP SP2 version. This new technology induced backward compatibility problems: some popular computer software does not work when the DEP is enabled, which is a non negligible problem for the wide adoption of such technologies. To bypass these new protection systems, instead of loading instructions in data memory that is no more executable, different intrusion techniques have converged to the so called Return-Oriented-Programming technique [218, 121]. After having identified a sequence of instructions already available into instruction memory, this technique makes the hijacked PC execute the preselected list of instructions. Then the Address Space Layout Randomization (ASLR) was created to counter such attacks, which basically consists in putting instructions at random locations so that the intruder could not predict and select them.

Concerning command injections in WebL, some languages provide libraries to filter the eventual instructions contained in input values like stated in [224], or more generally gives the user the possibility of configuring whether he wants to execute the incorporated script or not like the NoScript plug-in in the Mozilla Firefox web-browser [14].

[11] http://pax.grsecurity.net/

[12] http://www.openbsd.org/security.html

[13] http://www.intel.com/cd/ids/developer/asmo-na/eng/149307.htm

[14] https://addons.mozilla.org/fr/firefox/addon/722

More generally, protection systems could be used to define at different granularities, the rights a subject has on objects. Indeed, if an intruder takes the control of one process without the rights to craft the packets to conduct a TCP SYN Flooding, a botware will have to obtain these rights by other means to conduct such attack. However protection systems involve two main issues. First implementing the strict rights each process can have is not scalable. Then, so that a software works correctly in a Protected System, some expertise is needed to identify the rights the latter needs when providing its functionalities: a Web browser will not work if it has not the rights to send packets in the Internet. Different operating systems implemented protection systems with different level of granularities, like Amoeba [179], Chorus [215] or Mach [241]. Moreover, a proposal for implementing such a system using Smart Cards has been done by our laboratory [112]. Here again, taking apart some isolated projects like GrSecurity[15] , a quite isolated version of the Linux operating system, the idea does not seem popular among widely deployed operating systems.

Finally, even if different solutions exist, the deployment of these different techniques does not seem to be applied, as most of the malware still efficiently use code injection techniques as their weapon of choice [141].

Considering that end-host software faults exist and cannot be tolerated, the following section presents different ways to to monitor suspicious programs.

1.2.2 Monitoring Suspicious Computer Programs

After being suspicious of some computer behaviour, and knowing that intrusions were possible, some administrators used to ask (and still ask) some experts to audit it. Here again, computer scientists started automatizing this process in 1980 and Intrusion Detection Systems (IDS) were created [144, 178, 211]. To sum up, we can classify Intrusion Detection techniques into two categories: misuse detection and anomaly detection [185]. Misuse detection aims at detecting known attack patterns [140, 156]. Anomaly detection models the observed users behaviour and considers any significant deviation as an attack. Anomaly detection is not specific to cyber-intrusion detection, it can be used in among others, fraud, medical anomaly, and textual anomaly detection [95]. For a survey of the latest findings of Internet anomaly detection, the reader is invited to read the detailed survey by Patcha and Park [196]. The main difficulty anomaly detection has to face is the characterisation of Internet user's normal behaviour. Indeed, such behaviours can be very different, and change very fast. That's one of the reasons why pragmatic people don't believe in anomaly-based approaches [122]. However, even if being the most efficient short-term strategy, misuse detection has two main drawbacks. First, if the diversity of attacks is too important, this strategy is not scalable. Last, this strategy can only be reactive. Therefore, current systems try to use ad-hoc combination of both [209].

Intrusion detection systems can then be classified according to the location where the monitoring software is installed:

Host IDSes in the same computer where intrusions are observed. Very precise information can be gathered on the behaviour of program or connected users. However, if the computer has already been infected, the malicious program could try to trick or uninstall the monitoring program.

Network IDSes in an interconnection point on the Internet. The software inspects the packets going through its location. Less information can be observed than in the host configuration, but much

[15] `www.grsecurity.net`

more computers can be inspected. Another challenge such equipment needs to face is real-time constraint. Indeed, packets can go very fast, sometimes faster than the detection algorithm.

1.2.2.1 Host IDS-es

One of the most deployed host Intrusion Detection Systems are antiviruses. Such antivirus systems extensively use misuse detection techniques for pragmatic reasons. The so called antivirus signature updates are therefore mandatory to have an up to date protection system. These solutions need however to face, together with the limitations of misuse techniques, a side effect of the expressiveness of von Neumann machines. Von Neumann machines can compute computer programs. Malware developers have here again used this functionality to counter misuse detection techniques. Polymorphism [180, 73] and metamorphism [82] make the task of such IDSes much more difficult. Polymorphic viruses, before propagating, use ad-hoc algorithms to produce a different version of the binary. This algorithm takes the malicious instructions to hide as a stream of bytes and apply a reversible transformation to them. A routine is then appended at the begining of the transformed stream. The latter applies the reverse transformation to the new block to get the initial one back, and then executes it. Metamorphic viruses first compute an abstract version of themselves, called their semantic. Then, they compute a sequence of instructions conducting to the same results, but being as different as possible (in term of the stream of bytes) from the previous version. Indeed, $1+2$ and $(1-1)*0/3+15/5$ produce the same results even if different. The theoretical results give bad news to the antivirus mechanisms based on static analysis. Indeed, polymorphic viruses static detection problem is NP hard [221], as the metamorphic one [82]. NP hard problems are problems that can *reasonably* be solved[16] using a non deterministic Turing machine, i.e a computer that can execute an arbitrary and non predefined number of Turing machines in parallel. Implementations of such computer do not exist yet, and therefore these problems cannot be nowadays pragmatically solved. Antiviruses together with network IDSes need therefore to use other techniques, like virus execution emulation, to counter such trends [209]. The problems antivirus or IDSes need to tackle are therefore complex, and solutions implementations expensive, reactive, and probably not scalable. It's not surprising that the results obtained by commercial products are quite bad [100, 101, 20]. However, detecting malware is only the first part of the problem. Antiviruses also need to remove the infected binary from the system, which is another tedious task. Here again, these software products show their limits [195].

Antivirus software is installed in end hosts. However, network administrators cannot say whether for example all users have an up to date signature database in their system. That's, among others, a reason why network intrusion detection systems appeared.

1.2.2.2 Network IDS-es

Similar misuse detection schemes have been applied to the detection of malicious network byte patterns, using static [90] or dynamic [202] analysis. To handle network attacks, Network Intrusion Detection Systems (NIDS) use the techniques proposed in the network measurement field. Network measurement aims at understanding the characteristics and dynamics of the stream of bytes going through one interconnection device. This recent field of research comes from a need network or Internet Service Provider (ISP) administrators have. Indeed, after having interconnected different Internet networks, evaluating their efficiency is not trivial. Even if some end users report very bad bandwidth performance, understanding its causes needs expert auditing. Measurement therefore aims at understanding Internet dynamics

[16] Using an acceptable amount of computing time on an affordable amount of computers.

to automate or at least ease the design, the deployment, and maintenance of interconnection device and networks. Detecting Internet traffic volume anomalies is therefore of great interest for ISP or network administrators at large. Indeed, they can have to face different undesirable situations. Let's say two ISPs make a deal to share half of the communications between two parts of the Internet, $N1 <-> N2$. Half of the traffic will go from $N1 - ISP1 - N2$, and the other half through $N1 - ISP2 - N2$. Both of the ISP buy the infrastructures to handle half of the network communications of users in $N1$ and $N2$. However, let's say a network engineer working at $ISP1$ does an error and makes the traffic from N1 go to ISP2 first. Here, $ISP2$ has to handle 100% of the traffic, over-sizing its capacity. Detecting this kind of changes as fast as possible is therefore mandatory for ISPs. The same way, the appearance of bandwidth consuming network attacks, or worms create similar problems. Therefore, reactive algorithms to detect network traffic volume anomalies have been designed. These algorithms need to face two main problems: the real-time constraint again and legitimate traffic increases, commonly named flash-crowds [146]. Two different strategies are commonly used in measurement: passive measurement, which bases its analysis upon observed packets, and active measurements, which sends probe packets to extract more information.

The real-time constraint comes from the fact that in Internet's core networks, like the one managed by ISP1 and ISP2, interconnection devices already use most of their computing capacity to forward packets at some 10 or 100 Gigabits per second [143]. Therefore, some algorithms choose to work on periodic or random samples of such byte streams [159]. Most of the work done in the detection of anomaly in traffic volume has studied the number of packets and bytes time series. The studies have proposed different statistical models to distinguish normal from abnormal traffic, using different signal processing techniques [159, 217, 242]. Active detection paradigms propose different algorithms to detect similar anomalies [238, 158].

However, some attacks like the naptha attack we mentioned earlier, need very little bandwidth and such algorithms will perform poorly there. Indeed, the anomaly does not concern the volume of the traffic, but relies on the information exchanged by specific packets. That's why other techniques use more specific data, like the information contained in different packet headers. Some studies propose techniques to detect ICMP, UDP floods or TCP SYN floods [87, 186]. Even if such techniques can detect the attack we mentioned earlier, network attacks evolve the same way as malware and malicious activity adapt themselves to the apparition of antiviruses. Indeed, as we said earlier, the denial of service of a particular site could be provoked by mimicking a flash crowd [205]. Therefore, the previously mentioned techniques that focused on distinguishing flash crowds from denial of service attacks are useless in this case. The solutions depicted in [239] propose an algorithm to learn the normal behaviour of a web server clients, enabling the distinction from malicious flood requests. This technique is based on the assumption that the attacker is unable to intercept and collect the HTTP requests a legitimate user sends to the target. Let's take the example of a botnet that records different web activity pattern of clients from infected computers, by sniffing the network packets it sends. This botnet could then replay such activity to conduct an attack while being considered as normal by previously mentioned algorithms. We therefore do think that all these techniques, even if useful for some network attacks, could do little against a quite evolved botnet. Hopefully, instead of detecting network attacks, network measurement has tried to detect botnets too. If we take another look at the Figure 3 of our introductory chapter, the traffic sent by a corrupted node is sufficiently different to distinguish it from non infected ones.

First botnet detection techniques are based on extrapolating the results obtained from the local study of a botware. The studies in [114, 124, 222] exploit the fact that some botware use different dialects of IRC as C&C protocol. Therefore, they reduce the problem of detecting bots to the detection of such

dialects. This method can be quite easily bypassed by designing a mapping from normal IRC commands, to a language expressing malicious actions. Rajab et al. first analyze some patterns extracted by the local analysis of some botware, namely DNS resolution and IRC protocols towards some destinations in the Internet in [54]. Then, they extrapolate such behaviour to identify nodes conducting similar activities on the Internet at large. Livadas et al. propose another solution in [167]. After selecting IRC flows from others, they use machine learning techniques to differentiate legitimate IRC flows from bot flows. This technique can be tricked by the mapping technique we mentioned before. The study in [168] generalizes this approach in a framework to perform similar actions on other C&C protocols. Some bots make extensive use of the DNS protocol to hide or migrate some of their C&C servers. That's why some research has focused its efforts in the monitoring of the DNS protocol [232]. This approach uses a training set containing legitimate and malicious queries, and then classifies hosts making known or unknown queries by a bayesian classifier. The need for a valid data-set proves that the technique is not sufficient. Other techniques are based on the assumption that bots perform synchronized and distributed actions according to commands sent by the same C&C channel. Hyunsang et al. try to group in [99] computer nodes performing similar and synchronized DNS requests, making the assumption that botware perform similar and synchronized DNS requests. A very specific solution is depicted in [79]. Binkley et al. try to detect nodes using IRC protocols, and among them selects the ones performing a network scan. A network scan is the most simple way one could use to discover services in the Internet. One just need to choose a subspace of all IP addresses, and try each address in this subspace. Some botware use this technique to discover vulnerable services, or in other words possible propagation targets. Bots are then discovered by correlating an IRC activity with a network scan. Gu et al. generalized this approach to two HTTP channels and other network anomalies like spam or malicious binary download [4]. In their latest work they generalized the idea to any C&C [129]. Their methods first group, by clustering techniques, network nodes according to the different network applications they use and by the attacks they perform. They finally correlate both clusters and consider network nodes to be in the same botnet if they are both in the same correlated groups. We think this novel approach can be criticized by two facts. First, their algorithm needs first an algorithm to efficiently detect network attacks in the Internet (this is needed by their algorithm). However, if we could know that a node is perpetrating an attack, then we could deduce that the node is malicious, and that it must be analyzed without trying to detect whether this node is part of a botnet. Moreover, to evade these algorithms, bot developers can try to randomize the delay at which they interpret some orders or use multiple C&C protocols to trick the first clustering algorithms.

Finally, all these botnet detection techniques need to have information on bots behaviour. Indeed, they all base their misuse or anomaly detection algorithms on malware behaviours. This enables such researchers to select proper observable and to create the necessary data-sets to tune, and evaluate their algorithms. Therefore, a previous observation or study of botware functioning is most of the time mandatory.

1.2.3 Observing Malicious Behaviours with End-Host Software

Aiming at gaining information on malicious activity on the Internet, some people started to design what are called honeypots.

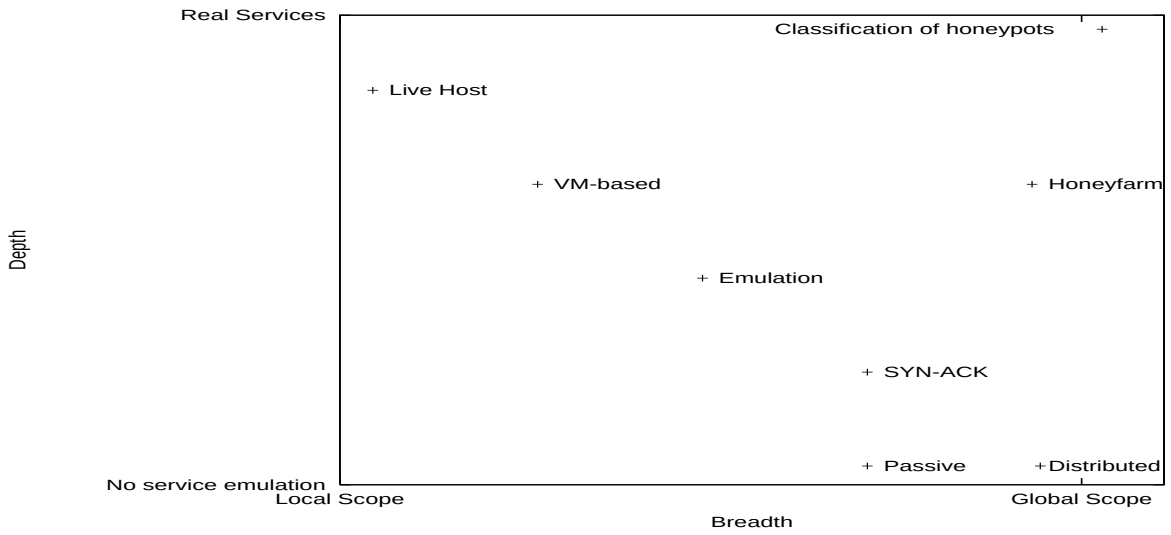


Figure 1.3: Honeypot classification

1.2.3.1 Honeypots

As stated by [119], there is no consensus on a global honeypot definition. In this thesis, we have chosen the one stated in [119]: "A honeypot consists in an environment where vulnerabilities have been deliberately introduced in order to observe intrusions."

As shown in figure 1.3, honeypots can be classified in function of two parameters [66]:

1. The interactivity they offer to the attacker, called Depth of the honeypot (Y-axis). Indeed, putting a real vulnerable server in a computer network presents two main drawbacks. First, extracting valuable information from a successful penetration is difficult to automatize. Then, hosting a deliberately corrupted node is hazardous and illegal. However, monitoring the requests sent to an unused IP address, without any interaction gives little information on the purpose of the source address: is it a malware trying to propagate ? Someone that misspelled an IP address ? A packet due to the bad configuration of a router ? That's why different level of service emulation have been proposed by different honeypots.

2. The scope they reach, called Breadth of the honeypot (X-axis). If we take the case of "silent" honeypots that don't make some kind of advertisement to attract attacks, they will only be targeted if they 'luckily' appear in some malware hit list. Therefore, putting a distributed honeypot network, with nodes disseminated around the Internet, has more probability to be in more hit-lists than a single one. However managing N honeypots, requires more resources.

The best interactivity/cost ratio has been achieved by VM-based, Honeyfarm, or Emulation solutions. VM-based solutions, are honeypots that use virtual machines [203]. A virtual machine can be seen as an abstraction of a computer machine, that runs on a real machine. Virtual machines offer very interesting features for honeypots. First, such machines enable to run multiple virtual machines on a single real computer. Then, such technology enables to take snapshots of machines. When taking a snapshot, all the state of the virtual computer is stored in the real computer. Then, by loading this state, the virtual

machine goes back to the state recorded and runs from there, as if nothing happened after having taken the snapshot. Moreover, snapshots enable to launch honeypots very fast (no more booting process). Finally, after analyzing successful intrusions, there is no need to clean the computer before reusing it, one just needs to load a clean snapshot. Virtual machines are therefore very useful to improve the automation and the deployment of honeypot experiments. With a fine-tuned use of this technology, Vrable et al. successfully deployed 2100 honeypots simultaneously using 10 real servers [233]. Having a virtual machine enables then to ease the implementation of modified von Neumann machines, used among other as honeypots. In [204], Portokalidis et al. used what they called memory tainting. Their system works by tagging all the bytes in memory storing information coming from the network. Then, they monitor the PC evolution, and consider an intrusion has been perpetrated if the PC points to a tagged byte. This solution has therefore the capability of automatically discovering new vulnerability exploitations by code injection attacks and generating vulnerability signatures. The very interesting work done by Costa et al. [107] generalizes this idea, and provides a global healing architecture. First, an algorithm is used to validate the generated signatures. A communication protocol propagates valid ones to other computer software presenting the same bug. Finally, the last algorithm automatically patches vulnerable software against this attack. The latter inserts a filter to the processing of input bytes to remove potential intrusions. This method has however a non negligible cost as the monitoring of PC evolution is expensive [235]. Instead of instrumenting a Virtual machine to implement a single honeypot, multiple honeypots can be instantiated by means of Honeyfarms.

Honeyfarms aim at resolving the problem of managing a network of distributed honeypots. This solution is based on two main technologies: network redirectors and virtual machines. Network redirectors process the requests between malware and honeypots' communications to send them to an appropriate and eventually dynamically created virtual machine acting as a honeypot. This approach aims at centralizing the equipment needed to host the honeypots, and limits the software to deploy in the Internet to the only redirectors. Xuxian et al. propose in [145] an architecture to develop honeyfarms, having vulnerabilities in server, and client software too, unlike the proposition done in [233] which only addresses server software. The last honeypot implementations aim to ease the deployment by emulating vulnerabilities.

Emulation based honeypots develop computer software emulating a vulnerable computer's behaviour. This enables to configure as wanted the interaction up to which the intrusion scenario is perpetrated. This approach started with a popular tool called Honeyd [206]. The level of interaction provided by the tool is low, and implementing vulnerabilities emulation software manually is a tedious task. In [162], Leita et al. proposed and developed the idea of synthesizing such software by mimic-ing vulnerable hosts. In [64], Baecher et al. developed the so-called vulnerability emulators with the specific goal of downloading malware trying to propagate. Once the malware is downloaded, the next step one is willing to make is to execute it. However, malware execution is the most hazardous task of a successful intrusion observation. Therefore, the automation of such observation needs special environments, i.e sandboxes.

1.2.3.2 Sandboxes or the Need for Dynamic Malware Analysis

In this section, we first describe the different techniques used by malware developers to make their detection more difficult. Then, taking into account that countering this new techniques is unfeasible using static analysis, we introduce the sandboxes that permit to automate the dynamic analysis of malware. These sandboxes have two purposes. First, compute a new version of the malware binary that can be statically analyzed. Then, the clustering of malware binary which helps reducing the number of malware to analyze.

A malware needs nowadays to perform its functionality in an environment where anti-malware technologies are installed. One of the most widespread defensive technology is implemented by means of antiviruses. An unknown malware will hardly be detected by antiviruses. However, a malware does not remain indefinitely unknown. For example, such a malware can be identified using computer forensics methods on a suspicious computer. Then, the malware is sent to antivirus companies so that malware reverse-engineering experts could add a new signature into their database. Therefore, to improve their lifetime, malware developers use different techniques to create obfuscated self-modifying code. Such techniques use combinations of metamorphism, polymorphism together with anti-malware systems' detection techniques to create *packers*: "programs that transform an executable binary into another form so that it is smaller and/or has a different appearance than the original to evade detection of signature-based anti-virus (AV) scanners" [130]. Quite a lot of packers are nowadays available [2, 47, 220, 69, 115, 174, 227]. In [130], Guo et al. state that there exists at least 2000 variants of packers, that could be classified into 200 different families[17]. As depicted in [137], the antivirus company Symantec received 499.811 malware in the second half of 2007, which makes a rate of 2738,68 malware to analyze per day. The malware analysis process needs therefore to automatize some tasks. However, it should be reminded that static analysis of poly and metamorphic binary is NP-hard. Given the proliferation of *packed* malware, dynamic analysis of malware binary seams to be mandatory. However, dynamic analysis implies to execute, or at least emulate, some parts of the malware which can be hazardous. That's why sandbox-es are developed to control the way malware binary is executed in the analysis process.

To ease the analysis of packed malware, *unpackers* have been designed. *Unpackers*, i.e programs that try to extract the un-obfuscated malware binaries, are sandboxes that most of un-obfuscated malware analysis methods rely on. The propositions depicted in [130] and [170] instrument variants of the Microsoft Windows operating system to monitor write and execute queries on the malware process address space, i.e. its memory. Depending of the proposed solutions, an additional intrusion detection system: an AV in [130] and a list of system calls considered as dangerous in [170], is used to analyze when modified pages are about to be executed. The unpacking is considered as finished upon the triggering of an alarm. The sandbox here is the software used to monitor the executions and modifications of the process address space. Instead of putting the sandbox into the kernel space of an operating system, other technologies use virtualization -or emulation- based approaches to isolate the modifications done by a malware to the system executing it. Tools like Anubis [1], CWSandbox [9], or Norman Sandbox [27] instrument an emulator and its operating system to perform dynamic malware analysis. Norman Sandbox it emulates distant servers too, trying to make a malware think it has Internet connectivity. This approach has been applied to some botware network communication dynamic analysis too [68, 176]. To limit such system's usefulness, some sandbox detection techniques, exhibit the differences between emulators and real computer's behaviour to detect whether the malware is executed in an emulator or not [230, 171], or even worse find exploitable vulnerabilities there [190]. The similar idea motivates the detection of emulated Internet environments [245]. Therefore, as depicted in [97], modern malware try to detect such kind of environment and modify their behaviour according to the execution environment. That's why Dinabur et al. propose in [113] to monitor similar observable as in [170], i.e. write/exec queries and system calls, but from a virtual machine hypervisor, in this case XEN [49]. Instead of using a computer software emulating a computer processor, such technology asks the processor to directly execute the instructions, and therefore limits the differences between a real and virtualized environment. Similarly, in [152], the monitoring process is installed in the virtual space too. However, this solution monitors system

[17] The different modifications of a same packers belonging to the same family.

state modifications (files creation, process creation) instead of system calls to create a causal dependency graph: which actions produced which modifications to characterize malware behaviour. Finally, dynamic analysis can detect malware trying to steal confidential information [116, 240], or provides a way to detect what events activate some sleeping malware [177].

However, the packing of malware is not the only cause of the number of different malware binaries antivirus companies needs to deal with. As depicted in [137] "most new malware samples are variants of previously-known samples through mutation of their source or binary code". Therefore, the clustering of malware is interesting because it can help reducing the amount of malware to analyze manually by grouping malware belonging to the same ancestor. To do so, Hu et al. propose to perform static analysis on a dynamically unpacked malware and group them according to the similarity of their function-call graph [137]. In [212] the information obtained from CWSandbox is used as clustering input. Similarly in [72] the clustering input is obtained from the Anubis sandbox. As shown by emulation detection techniques used by malware, the interaction level of the sandbox influences the information that can be extracted from the dynamic analysis of malware. Previous solutions did not offer real Internet interaction to the malware being analyzed. Indeed, this can be hazardous, as a malware could try to attack a third-party APS during an analysis. That's why in [65], a solution based on the clustering of information extracted from the sandbox depicted in [152], the author state that "the [sandbox] is partially firewalled so that the external impact on any immediate attack behaviours (e.g. scanning, DDoS, and spam) is minimized during the limited execution period". The lack of precision[18] on the solution adopted to handle such a big problem shows that monitoring dynamically analyzed malware's network communications is open for contribution. Indeed, botware execute actions according to the orders they receive from the C&C communication protocol. Therefore, if the sandbox does not let the malware communicate through this protocol, the malware would not perpetrate some actions. However, letting a malware perpetrate intrusion or network attacks towards remote systems is illegal in most developed countries. Therefore, nowadays botware dynamic analysis is performed using ad-hoc sandboxes, where, as we have previously said, little details are given about the policy used to monitor their network behaviour [135, 98, 128], or where no inspection is performed on the communications deliberately sent to the Internet [1, 9].

We have proved in this section that significant advances have been done in correct software development. Given the fact that such techniques are not universally adopted, other techniques exist to limit the side effect of successful intrusions. However, such solutions have not been adopted yet and therefore, an heterogeneous set of algorithms and computer software have been developed to monitor suspicious software. Finally, to gather information about Internet's malicious activity, another kind of software aiming to observe malicious activity has been developed too. We'd like to emphasize again that both monitoring and observation software we are aware of have not taken into consideration the advances done in correct software development leading to possible strange situations.

1.3 Monitoring Malicious Behaviours using Correct Software in Midpoint Devices

We do think that the problems raised by botnets are due to the underestimation of the challenges a software connected to the Internet needs to face. The recent critical vulnerabilities found in the TCP stack of

[18] The reader is invited to imagine what a partial firewall could mean, and to what extent it could minimize which impact on third party ADPS.

Microsoft Windows operating system families, MS09-048[19], or in the SCTP stack of the Linux kernel, CVE-2009-0065[20], argue for this idea. As we stated earlier the race for innovation influences the development process to create new potentially incorrect functionalities, and therefore the apparition of critical bugs. The results obtained by the OpenBSD[21] project shows that such a race does not explain it all. Quoting the home page, "[their] efforts emphasize portability, standardization, correctness, proactive security and integrated cryptography". Their development process that relies on heavy expert human auditing of software developed in C language, has still lead to 2 critical vulnerabilities during 15 years of existence. The latter experiment proves the point of John Backus that criticises von Neumann inspired languages. For these reasons, we state that endhost software cannot reasonably be trusted, and that they must be monitored. Given the fact that once an intrusion is successfully perpetrated on a computer system, due to the lack of intrusion tolerant systems, most of the software connected to this system can be controlled by the intruder. Therefore, we state that the monitoring software should be installed in interconnection points, what we call midpoints. Given the analysis we made on the source of the problems raised by botnets, our monitoring software must be correct. However, as we depicted earlier, formal verification of an entire software is nowadays still too expensive. Meanwhile, statically type-safe languages that use garbage collectors offer, even if not total correctness, at least guarantees against code injection threats, like stack and heap overflows, or format string vulnerabilities. Moreover vulnerabilities come from bugs. Considering that such languages provide a static bug checking algorithm, the developped system will be safer using such a language. We have therefore chosen for pragmatic reasons to formally prove only the main parts of our monitoring software installed in a midpoint, and delegate the checking of the correction implementation to a static type checking algorithm. The following chapter describes the details of this tool we call midpoint inspection device.

[19] http://www.microsoft.com/technet/security/bulletin/ms09-048.mspx
[20] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0065
[21] `http://www.openbsd.org/`

2

Motivations, Design and Implementation of a Midpoint Inspection Device

> "Knowledge is only part of understanding. Genuine understanding comes from hands-on experience". Seymour Papert.

To understand the functioning of our tool, the first section makes an introduction on how endhosts in the Internet inter-operate. The next section illustrates then how our midpoint inspection device incorporate these inter-operable mechanisms to provide an expressive language and algorithm to configure this inspection. The interpreter of this language gives the possibilities to make with a single software, what up to know required multiple ones. Finally, the last section describes the benchmarking of our midpoint inspection device on two end-host monitoring case studies.

2.1 Challenges Faced by a Midpoint Device

2.1.1 Midpoint vs Endpoint

We describe here the difference between midpoint and endpoints' point of view. First, the layering principle is described so that the reader understands how endpoints in the Internet communicate.

2.1.1.1 The Layering Principle Used by Endpoints

The midpoint inspection device needs to perform inspections concerning the packets seen between monitored endpoints. These inspections aim at making or validating that endpoints behave as expected. There are however some major issues such a midpoint device needs to deal with.

The Internet endhost functioning is deeply inspired by the layering principle of the *Open Systems Interconnection* (OSI) model [244]: "The basic idea of layering is that each layer adds value to services provided by the set of lower layers in such a way that the highest layer is offered the set of services needed to run distributed applications. Layering thus divides the total problem into smaller pieces". Let's try to understand why such layering is interesting by analyzing the trips of a Chinese and American diplomats going to an international conference in Moscow.

The Chinese diplomat first takes a car to reach the Beijing Capital International Airport. From the airport he flies to Russia, and then takes another car to reach the conference room. To do so, he first needs to know how to drive a car according to Chinese rules of the road. He then takes a seat on the plane and lands at Moscow. From there, he needs to drive a car according to Russian rules. Analogous knowledge is required from the American diplomat. What should be noticed here, is that both diplomats

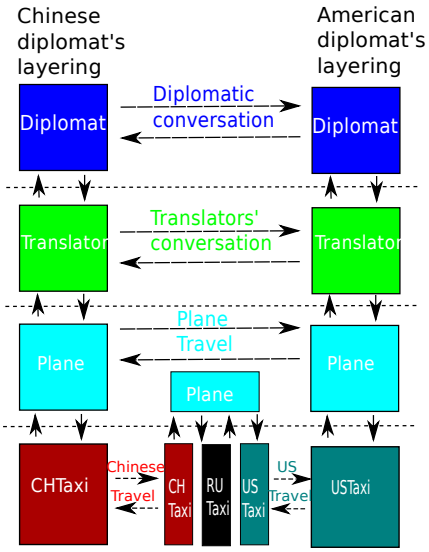


Figure 2.1: Different layerings of Chinese and American diplomats

do not need to learn how to drive in Russia. Indeed if they agree on a meeting point at Moscow, only one diplomat has to learn how to drive on Russian roads when the other is the passenger. Going further, let's say these diplomats go to conferences all around the world. Taking the time to learn all the road rules will non negligibly impact the remaining time for doing their diplomatic work. Therefore, the layering approach proposes to use local taxi drivers knowing how to drive all around their local part of the globe. Our diplomats are now in Moscow.

Once in the conference room, to debate, both diplomats need then to understand themselves. Here again, instead of learning others' languages, the layering principle will use translators from a common language to their respective languages. These different layerings are depicted in Figure 2.1. Layering improves the factorization of tasks, while providing flexibility. However making each layer cooperate has a cost. If for example our Chinese and American diplomats know Esperanto, they can directly talk to each other without asking their translator service, but they'll have difficulties talking to all the other non Esperanto speaking diplomats. The tradeoff between genericity and performance is finally one of the main concerns an Internet system designer has to balance [34]. For the ones wondering how a client connected to a wifi hotspot connects to a Web server wired to the Internet, the analogy we made could be mapped. Even if the way the different services interoperate is not exactly the same, the layering proposed by the TCP/IP model is depicted in Figure 2.2.

From now on, we will note the different layerings used by Internet applications using the following notation: $Wifi/IPv4/TCP/HTTP$. We will call *composition*, a superposition of two protocol stacks, and describe it by means of the $'/'$ symbol.

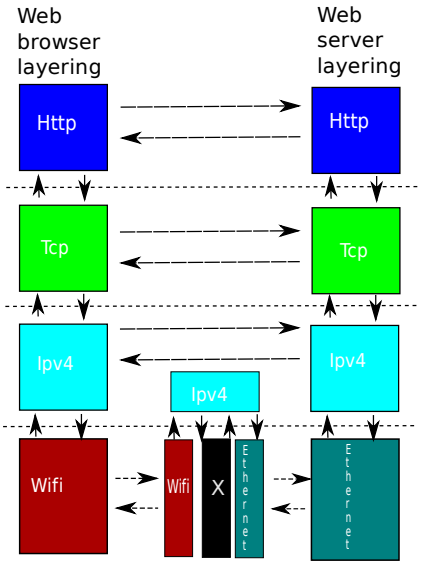


Figure 2.2: Tcp/Ip model applied to Web

Therefore Internet endpoints' behaviors depend on their layerings and on the implementation of each layer. Going back to our analogy, the community of diplomats in the conference at Moscow has agreed to stop taking planes, and to use the Internet to make virtual conferences.

Both diplomats will go to local translators' offices. Then, by using a computer interconnected to the Internet, where a software to make video conferences has been installed, all the translators around the world will connect to the virtual conference as depicted in Figure 2.3. To transfer the images and sounds of translators' offices, the video conference software will use some layering over a protocol offering the service to travel around the Internet. Nowadays the $IPv4$ protocol is the 'plane' most of layerings choose to take. This protocol offers what should be seen as an unreliable postal service. When a protocol stack $S1$ is composed over $IPv4$, i.e $IPv4/S1$, $S1$ could put its own envelop of size at most the maximum size an $IPv4$ packet can carry, called Maximum Transmission Unit MTU. Then, it writes in the address field of the envelop the destination address of the message and sends it to the Internet. The Internet ensures, like the cheapest postal service, to do its best to send an unaltered version of this envelop. However, there are quite a lot of challenges to overcome before being able to make a videoconference in the Internet. The RFC 3439 [34] mentions the following:

- Error control: check whether packets have been altered during the transfer of information. Indeed, the Internet does not guarantee that the bytes it transfers will not be modified. For example a sender could send $hello$ in an $IPv4$ envelop, and the destination receive $hells$.

- Flow control: check at what rate a receiver can receive a data. The same way we ask people talking too fast to slow down, computers need to negotiate a conversation pace too.

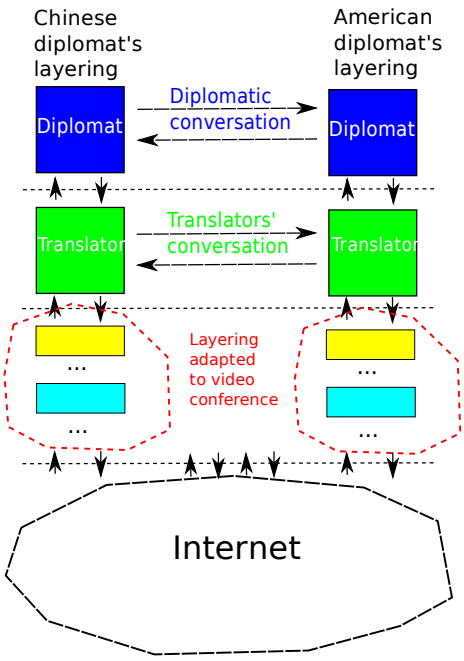


Figure 2.3: Video conference layering.

- Fragmentation: if the message $S1$ wants to send is too big to fit in a single $IPv4$ packet, the message should be fragmented in multiple smaller messages, sent to the Internet, and then reassembled at the endhost.
- Packet loss, reordering, and duplication control. As stated in [143], Internet can drop, duplicate or reorder a sequence of packets. For example a sender could send $hello$, and the endpoint receive ole, by loosing the h and the l, and swapping the remaining e and o.
- Connection setup: the same way two distant people willing to discuss an issue should first agree on a meeting time and location, some communications need to establish a connection before chatting.
- Multiplexing: to reduce the number of needed translators, videoconference software, and interconnected computers, two Chinese diplomats should go to the same translator's office. In this case the translator must provide a mechanism to know which message is addressed to who.
- Addressing/Naming: each entity should be identified, like diplomats, translators, or computers.

However, new applications like for instance video conferences will probably have to face problems previous applications had not. We can mention these two related problems:

- Delay/Jitter management: the delay between two endhosts is the time to go from one point to the other. The jitter measures the variation of the delay. These two parameters greatly influence the quality of a video conference. Indeed, the receiver will not see the face of its correspondent before a time related to the delay of the first packet. The same way the jitter influences the time

at which the other images/sounds arrive. For the people used to watch streamed multimedia data[1], an increase of the jitter may induce a pause on the displaying of an already started movie.

- Congestion control: endhosts know that the pipes they are using to communicate are shared among a lot of different other endhosts. All the endhosts should therefore responsibly use these limited resources to improve the overall use. More pragmatically, let's say a pipe shared among 10 endhosts, can at most send 10 packets per second. One endhosts sending 8 pkt/s while others are quiet is fine. However when the 10 end hosts start sending 2 pkt/s, the pipe will drop half of the packets. Endhosts should then decide to reduce their sending rate to 1pkt/s. Of course, this problem is strongly correlated with the delay/jitter management.

By extrapolating this trend, new applications will probably need to face new problems. The way Internet system designers tackle these unforeseen problems is proposing new sets of protocol stack layerings, and developing new protocol stacks when the layering of existing ones cannot address critical problems. Finally the packets sent in the Internet look like Russian dolls, or said otherwise, a succession of envelops *encapsulated* into other envelops. Endpoints behave according the interaction between different protocol layers. The following section aims at showing why the computations made by endhost layerings cannot be used to monitor endhost communications from a midpoint device.

2.1.1.2 The Undeterminism Faced by Midpoints

To illustrate the computations midpoints need to perform, let's take the case of a firewall, whose policy states that no messages should rise diplomatic conflicts. To do so, the firewall has to analyze the messages in $S1$, extract the messages from $S2$, analyze translated messages, and verify that the diplomats do not send insults. One could say that the firewall could just install the software needed to implement the layering $S1/S2/Translator$ in the midpoint device, and implement another stack $NoDiplomaticConflict$ to compose over $S1/S2/Translator$. We show why this proposition is incorrect with two different examples.

A first problem raises with fragmentation. Let us say $S2$ provides a fragmentation service to $Tanslator$. Let us say one Translator asks $S2$ to send the message $'abcd'$. Because of some constraints, $S2$ needs to split this message into three different parts $'a'$, $'bc'$ and $'d'$. Considering that the Internet can drop or reorder some messages, and taking into account that $S1$ does not offer a reliable communication pipe, even if messages are sent in the order $'a'$,$'bc'$ and $'d'$, the $S2$ in the receiver endpoint can receive $'bc'$,$'a'$,$'d'$. Therefore $S2$ needs to find a way to express that $'a'$ comes first, that $'bc'$ comes next and that $'d'$ ends the message. That's why $S2$ chooses to index each letters of the messages so that each messages are correctly reassembled, i.e. $S2$ actually sends $('a',0)$, $('bc',1)$, $('d',3)$. Let us say the receiver $S2$ has first received $('a',0)$ and $('d',3)$.

In a first case, let us consider the case where the receiver $S2$ gets $('bc',1)$. The reassembly algorithm computes that $'abcd'$ is a a contiguous stream of 4 bytes starting at index 0. Therefore $S2$ will reassemble the three fragments $[('a',0),('bc,1)',('d',3)]$ and send $'abcd'$ to $Translator$.

In a second case, let us say that instead of $('bc',1)$, the message $('zbc',0)$ is received. Indeed, even if a priori abnormal, no one prevents one endhost to send $('a',0)$ $('d',3)$ and $('zbc',0)$ messages. Indeed this behaviour can be motivated by malicious purposes or not[2]. Putting the causes of such situation aside, the $'z'$ and the previously received $'a'$ have the same index, i.e. 0. Therefore both $'abcd'$ and

[1] Like a video in Youtube.

[2] For example this could be a way to create an entropy generator based on the reordering rate of an Internet link.

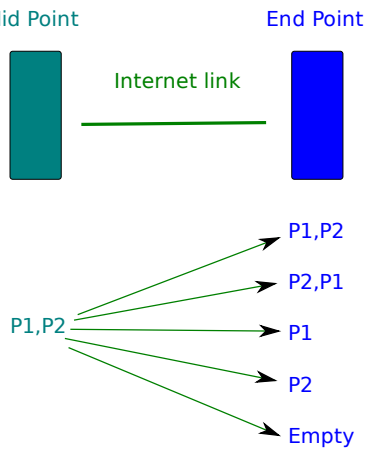


Figure 2.4: Difference between the sequence of midpoint and endpoint packets

$'zbcd'$ represent a contiguous stream of 4 bytes starting at the index 0. If the algorithm to perform the reassembly is standardized and states that the first received bytes should be chosen, then *abcd* would be sent to Translators (as $'a'$ has been received before $'z'$). If not, each $S2$ implementation will do as it wants. This creates another problem for the midpoint inspection device as it does not know how the endhost will behave or in other words, whether $'abcd'$ or $'zbcd'$ will be reassembled and sent to the receiver endhost's $Translator$. The detection of possible conflicts is in this case undeterministic.

The drops and reorderings induced by the Internet creates another source of undeterminism too. As depicted in Figure 2.4[3], when a midpoint inspection device sees and forwards the sequence of packets $(P1, P2)$, it cannot assume that the endhost will receive $(P1, P2)$, as Internet could drop or reorder packets.

The job of a firewall, and more generally of midpoint inspection devices, is therefore to monitor network messages produced by different layerings, and implement a policy taking into account the undeterminisms raised by being a midpoint device. We present in the following section how existing solutions handle these problems.

2.1.2 State of the Art of Midpoint Inspectors

The midpoint inspectors have been up to now designed focusing on pragmatic point of views. First, a big attention has been given to the performances of the inspection. Indeed such computations have to face the real-time constraint imposed by the rate at which the inspected network packets arrive. When a second packet comes before the firewall finishes to process a first one, the firewall needs to buffer it. Therefore

[3]Duplication of packets are not taken into account.

this buffer grows each time the firewall is too slow, and as its size is limited, a not fast enough inspector will not be able to process all the incoming packets. Packet filters have for example tried to address these performance problems. As stated in [74] "A packet filter is a programmable selection criterion for classifying or selecting packets from a packet stream in a generic, reusable fashion.". A packet filter put in a middle-box can therefore be used to implement a programable firewall. The packet filters compilers like [74, 85, 237], have targeted high speed networks. However, for performance reasons, they choose not to address, neither in their current implementation, nor in the possibilities given by their language, *TCP stream normalization* [234]. However, taking into account that the great majority of the layering to inspect uses the TCP protocol, a special attention has been given to its inspection too.

The TCP protocol offers a fragmentation service to its upper layers. Taking into account that malicious users can produce the situation depicted in Section 2.1.1.2 , the *TCP stream normalization* chooses to drop all the ambiguous packets. Therefore as they cannot perform this normalization, these packet filter compilers cannot reliably inspect the protocols over *TCP*. This problem has been addressed by Intrusion Detection [197, 90] and the deduced Prevention systems [243] like the ones depicted in [43, 147, 172]. However these systems aim at detecting attacks and therefore don't offer the possibility to configure a policy independent of attacks or intrusions.

The language described in [84] provides an application level protocol analyzer that could be used to develop an application level firewall. At least two critics can be made to this work. First this language does not offer the possibility to parameterize TCP or UDP protocols inspections. However, some inspections can require to check that endhosts perform a specific congestion control algorithm, while others do not care about it and therefore do not need to "waste" computational resources on such checking. Then, this language makes multiplexing according to the indexes used by the TCP protocol: *source* and *destination ports*. This solution does not address the problem arose when two endhosts use a pre-negotiated non standard port.

The midpoint inspectors are driven by configuration languages. The work done by Liu et. al. addresses the problem of stateless firewall configuration analysis. Stateless firewalls are firewalls that filter messages without taking into account previous messages. In [126, 165, 166], the authors propose a tool suite to help firewall administrators design consistent (the sequence of generated firewall rules implements the administrator policy), complete (all possible packets are taken into account), and compact (there is no redundant rule) firewall rules. Stateless firewalls are however limited, as they cannot perform stateful inspection. For example checking a correct TCP three-way handshake is performed or applying *TCP stream normalization* is unfeasible by a stateless analysis. In [125], a modeling of stateful firewalls as a sequence of static and stateful rules is proposed. This model is inspired by the syntax of common stateful firewalls [29, 26] where the statefulness information is encoded in boolean variables. The language proposes then to filter communications according to values of the current packet together with the values of related state variables, which are updated by the firewall itself. A similar study proposes to model firewall rules using a specific ACL language, to check the validity of a given firewall. First the firewall configuration language is modelled in this language. Then their interpreter generates automatically some firewall policies, and verifies their correction using automated live testings [55]. Finally, in the quest for genericity, a high level packet filter language is described in [70]. However, this solution only uses, like all the midpoint inspection devices we know, *monolithic protocol layerings*. By this idea, we express the fact that such devices are based on the configuration of a set of predefined layerings $SL = \{S11/.../S1j_1, ..., SN1/.../SNj_N\}$. Therefore if a policy asks to implement an inspection corresponding to a layering $L \notin SL$, a new midpoint inspection device should be built. We see three different possible causes for this problem:

1. L contains at least one protocol stack not implemented by any layering in SL. For example $L = IPv4/SCTP/HTTP$ and $SL = \{IPv4/TCP/HTTP, IPv4/TCP/FTP, IPv4/UDP/DNS\}$. There is no magic solution for this problem, all firewalls will need to implement $SCTP$ to process such a layering.

2. L is a layering containing protocol stacks already implemented in the firewall, but not in this order. We emphasize two reasons leading to such a situation.

   - L asks to perform a shorter version of a layering. For example if we take $L = IPv4$, all the layerings in SL that have $IPv4$ as first element, i.e. $IPv4/OverIPv4One/..., IPv4/OverIPv4Two/..., ...$, will at least unnecessarily extract the encapsulated message in the $IPv4$ packet. Indeed the inspection to perform is just based on information contained in $IPv4$, the computations done to check $OverIpv4One$ and $OverIpv4Two$ are useless. This fact has created the distinction between standard firewalls [26, 29], which use $IPv4/TCP$ or $IPv4/UDP$ layering, and specialized application level firewall or IPS [147, 103], doing layerings like $IPv4/TCP/HTTP$. From our point of view, a single device should be able to configure both inspections.
   - L asks to perform a longer version of a layering. Let's take the case where there is a cycle in one of the valid layerings, like in $Eth/IPv4/GRE/IPv4/GRE/IPv4/TCP/HTTP$, or more generally $LStart/S_x/LCycle/S_x/LEnd$. Therefore the set of predefined layerings being finite, they cannot inspect layerings like $LStart/(S_x/LCycle)^N/S_x/LEnd$, N being the length of the longest layering in SL. Tunneling protocols, among others, provide a way to create such cycles. If we refer to the manual of the Snort IPS, layerings like $Eth/IPv4/GRE/IPv4/GRE/IPv4/TCP/Payload$[4] cannot be handled even if a single encapsulation can be used. In other words, inside Snort, the implementation of the GRE protocol inspection cannot be reused.

Indeed, handling *monolithic protocol layerings* is the main limitation of all the midpoint inspection devices we know. The research done in network protocol architectures has lad to what are called, dynamically configurable protocol layerings. Such solutions split one protocol into smaller primitive building blocks, splitting the complexity among smaller problems. In other words such architectures apply the layering principle to one protocol. Different implementations of such solutions exist like Coyote [76], Cactus [77], x-kernel [138], APPIA [201] or ETP [118], the one developed in our laboratory. To sum up, instead of having a layering like $S1/S2/S3$, such protocols will split $S2$, into a set of different layerings providing the set $\{S1/\{s2_{11}/.../s2_{1n_1}, ..., s2_{z1}/.../s2_{zn_z}\}/S3\}$ instead of $S1/S2/S3$. The aim of such architecture is to choose among these different layerings the best suited to one situation. In this case, the probability of getting layering cycles is greater. Moreover the development effort needed to monolithically implement all the layerings is not scalable: the task of implementing all the possible branchings becomes as tedious as developing a new protocol inspector.

The idea described by the language in the FFPF packet filter [85], allows different filters to be composed. This is up to our knowledge the only solution to give the user the capability to configure a non predefined set of layerings. However their solution does not provide algorithms to check neither the correctness of the configured layering, nor the factorization of unnecessarily duplicated computing. The next section describes the model and implementations of the solutions we proposed to these problems.

[4] http://www.snort.org/docs/snort_htmanuals/htmanual_2832/node35.html

2.2 Composing and Parallelizing Midpoint Inspection Devices

To sum up, we have enumerated three main problems:

1. midpoint inspection devices should perform an inspection of messages being in the middle of endhosts, and then cope with losses, reorderings, ...
2. related to this problem, midpoint inspection devices do not know all the pre-negotiated informations between endhosts,
3. endhosts can use protocol layerings that are impossible to analyze with *monolithic* approaches.

Going back to the tradeoff between genericity and performance, we have first focused our design on the genericity.

2.2.1 Definition of the Midpoint Inspection Device's Configuration Language

If we enumerate the functioning of the different midpoint devices used in the Internet, we have:

1. Network IDSes: midpoint devices that analyze endhost communications and rise alarms upon attack detection.
2. Network IPSes: midpoint devices that analyze endhost communications and drop packets detected as part of an attack.
3. Application Detection: midpoint devices that analyze endhost communications and identify the highest protocols used by the different communication layerings [114].
4. Shapers: midpoint devices that analyze endhost communications and police the bandwidth used by different protocol layerings.
5. Firewalls: midpoint devices that analyze endhost communications and drop packets outside a given policy.
6. New devices: midpoint devices analyzing endhost communications and taking decisions unpredicted by nowadays computer scientists.

There is quite a lot of common points among all these devices, and differ mostly on:

- The decisions they take. IDSes rise *alarms*, IPS es, Shapers, and Firewalls *accepts,drops*, Application Detection *identifies*,
- The reason why they take their decisions: IDS and IPSes because of attacks, Shapers because of bandwidth constraints, Application Detection, because of used application protocols and Firewalls because of policies.

Finally the following definition cover all these devices:

Definition 1 *We call midpoint inspection device a midpoint device that analyzes endhost communications to take decisions according to a policy.*

Let *DS* be the set of such decisions. The functioning of these endhosts is driven by the use of different protocol layerings, like $S1/S2/Translator/Diplomat1$, $S1/S2/Translator/Diplomat2$. Policing the messages exchanged between the different layers enables to make endhosts behave as expected. Therefore the policy concerns three different scopes:

Layer scope: what decisions should be taken when seeing the messages at some layer.

Layering scope: what decision should be taken from the interaction between the different layers.

Set of layerings scope: what decision should be taken from the interaction between the different layerings.

Instead of proposing the configuration of a predefined set of layerings, the user implements the policy he wants by means of our configuration language. The layer scope is implemented by what we call *midpoint inspectors* (MI). The syntax we use to configure an MI follows the pattern *ident* $(\{argc\{,argc\})+$, with $argc ::= arg\_key = arg\_val$, using an EBNF-like[5] notation. That way we can produce instructions with variable number of arguments. To configure a given policy at a layering, the / operator is used to compose the different MIs. Then to create layerings, we use the parallelization | operator. Finally [and] are used to modify the precedence of | and /, i.e $S1/S2/Translator1|Translator2 \neq S1/S2/[Translator1|Translator2]$. Let us say two diplomats one from the Republic of Congo speaking Lingala, the other from Hungary speaking Hungarian, are talking to each other using the Internet by the help of translators. Let us say the following policy must be implemented:

- Messages in Hungarian and Lingala are accepted.
- When an insult is said, a report should be sent to the government of the diplomat insulting the other.

The following rule could be used to implement such a policy:
$S1/S2/[$
$Translator(input\_language = Lingala, output\_language = Hungarian)/CheckInsult(report\_to = RepublicOfCongoGov)|$
$Translator(input\_language = Hungarian, output\_language = Lingala)/CheckInsult(report\_to = HungaryGov)]$.

If a message is sent for example in Italian, the message will be dropped. Our tool aims at helping building a fault tolerant Internet by implementing a Protection System for Internet networks. Therefore this drop decision comes from the system closure principle (Section 1.2.1): no action is permissible unless explicitly authorized. Changing this "default" decision, even if very easily implementable, should be done very conscientiously, as it drastically changes the purpose of the tool.

If the policy states:

- Messages in Hungarian and Lingala are accepted.
- When an insult is said by the Hungarian or Congolese diplomat, a report should be send to the government of the diplomat sending the insult.
- Conversations in other languages are accepted.

[5] Extended Backus-Naur Form.

By adding the *MI AllowLanguage*, that accepts all the messages sent in the languages described by the *language* parameter, the following rule could be used:
$S1/S2/[$
$Translator(input\_language = Lingala, output\_language = Hungarian)/CheckInsult(report\_to = RepublicOfCongoGov)|$
$Translator(input\_language = Hungarian, output\_language = Lingala)/CheckInsult(report\_to = HungaryGov)|$
$AllowLanguage(language = not(Lingala\ or\ Hungarian))]$.

The parsing of the configuration language can be easily achieved by means of grammatical and lexical analyzers, produced by tools like Yacc, and Lex among others [164]. Then, we should check whether the written rule is valid. If we go back to the layerings depicted when illustrating the fragmentation problem (Section 2.1.1.2), *Translator* takes messages reassembled by $S2$, which can be different from messages sent by $S1$ to $S2$. Therefore, even if $S1/S2/Translator$ is valid, $S1/Translator$ is invalid. Indeed, *Translator* analyzes reassembled messages output by $S2$ when $S1$ outputs non reassembled messages. Finally, we should synthesize an optimized version of an algorithm to take decisions. After having solved these last two problems, we create a midpoint inspection device interpreter. The same way higher level languages use an instruction set of von Neumann Machines together with parameters to configure their computations, we use a predefined set of MIs to interpret our decisions' computation.

2.2.2 Model of the Midpoint Inspection Device

We consider that the static type-safety of our algorithms' implementation is *mandatory*. The Hindley-Milner static type-safety inference algorithm has been first designed for the *ML* language [175]. Therefore, we choose to present these algorithms using a dialect of this language. The dialect we use is simple enough to be easily translated to other statically type-safe languages, like CamlLight [7], SML [44], F# [12], Haskell [13] or OCaml [28] the one we used to implement the prototype. The proof of our algorithms are given in the more standard style of von Neumann [59].

First, we need to make a very short introduction to this dialect.

2.2.2.1 *ML* dialect

To create a tomato type,

```
type tomato
```

is written. To say that potato and tomato are vegetables, we use variant types:

```
type vegetable = T of tomato | P of potato
```

For the ones familiar with the C language, a similar C program is:

```
/* assuming the types tomato and potato have been created */
typedef union {
   tomato tom;
   potato pot; } _vegetable;
typedef enum { T,P } v_type;
typedef struct {
   v_type t;
   _vegetable v;
} vegetable;
```

The definition of a variant type creates associated *projection* and *injection* operators. Indeed when processing a vegetable l, the projection of a variant type to its specific type is done this way:

```
match l with
        | T tom -> (* process tom, a variable of type tomato *)
        | P pot -> (* process pot, a variable of type potato *)
```

Here again, the translation to C can be:

```
  switch (l.t) {
  case T:
  /* process l.v.tom, a variable of type tomato */
    break;
  case P:
  /* process l.v.pot, a variable of type potato */
    break;
  default:
    /* should not be here */
    break;
  }
```

To create a vegetable from a tomato *tom* the *injection* operation from tomato to vegetable is done by *T tom*. The $*$ operator is used to create tuples. The $->$ token is used to express functions type too. For example the prototype of a function taking a tomato and returning a tomato is written like that:

```
val new_tomato: tomato -> tomato
```

The prototype and body of a function creating a pair of vegetables from two arguments, a tomato and a potato, are written like this:

```
val create_tomato_potato: tomato -> potato -> vegetable * vegetable
(* body of create_tomato_potato *)
let create_tomato_potato tom pot = (T tom,P pot)
```

To define polymorphic types, i.e. types whose definition depends on other types, the "single quote" ($'$) token is used. For example after creating a polymorphic list, the creation of a vegetable list is straightforward.

```
(*the list contains values of the parametric type 'a*)
type 'a list = NonEmpty of ('a * 'a list) | Empty
type vgtbl_list = vegetable list
```

To create a recursive function, the "rec" token must be used like that:

```
(* present l a, returns the boolean: a is in l *)
let rec present l elm =
  match l with
    | NonEmpty (hd,tl) ->
      if hd = elm then true
      else present tl elm
    | Empty -> false
```

We can now describe how we model and implement the midpoint inspection device.

2.2.2.2 Details of the Midpoint Inspection Device

First the decision set *DS* should be implemented. We implement it by a type.

```
type ds
```

MIs implement the decisions that should be taken when seeing the messages at a given layer. We use a type to implement the messages of this layer. Then, to model the statefulness of the MI, its state is implemented by another type. Finally, an MI could want to send a message, like once a fragmented message is reassembled, to the MIs in the next layers. Therefore, the messages output by MIs must have a message type too. Before going further we can now implement the genmsg type containing all the different messages:

```
type genmsg = Msg_1 of msg_layer_1 | .. | Msg_n of msg_layer_n
exception ProjectionError
...
(* for all msg_layers, or in other words for all i in [1,n] *)
let genmsg_from_msg_layer_i msgi = Msg_i msgi (* injection *)
let genmsg_to_msg_layer_i gmsg = match gmsg with
    | Msg_i msgi -> msgi (*projection*)
    | _ -> raise ProjectionError
...
```

The algorithm implementing the Layer scope, i.e. the one returning the decisions taken by the MI, is implemented by the use of a function. Each MI has two message types in the set:

MSGSET={msg_layer_1 ,... , msg_layer_n}

The first message type is used as *input* message, or argument of the inspection function. The other is a possible *output*. To keep a track of the correspondence between the *input* and *output* message types of the MI and the element in MSGSET, we use two surjections[6]. : *imi* like input of MI and *omi* like output of MI. Let MISET be the set of MIs. Both surjections are defined in $[1,|MISET|=J] \rightarrow [1,|MSGSET|=n]$. For example msg_layer_<imi[1]> represents the input message type of mi_1. To sum up the MI mi_j are defined by six elements.

```
type msg_layer_mi_j = msg_layer_<imi[j]>
type msg_uplayer_mi_j = msg_layer_<omi[j]>
type state_mi_j
type outinspector_mi_j = DS_j of ds | Out_j of msg_uplayer_mi_j
type inspection_result_mi_j = state_mi_j * outinspector_mi_j
val inspect_mi_j: state_mi_j -> msg_layer_mi_j -> inspection_result_mi_j
```

What should be noticed here, is that the inspector function returns the updated state together with either a decision *or* a new message to send to the next layers mentioned in the rule. Two remarks can balance this design choice:

- Why sending a message to the upper layers is not considered as a decision ? The communication of messages between all the MIs, and most precisely the interoperability of these communications is the most important part of our system, or at least the biggest source of bugs. Therefore we have chosen to impose the way each MI communicates with each other, by not considering the sending of a message to the upper layer as a decision, to have a way to ensure the validity of communications.

[6] $f : A \rightarrow B$ is surjective $\Leftrightarrow \forall y \in B, \exists x \in A, f(x) = y$

- Why an MI cannot take a decision *and* send a message to its upper layers ? Indeed, a policy would like to know all the alarms raised by all different layers to, for example, have en evaluation of the different violations done by a network communication. For example in a layering like $TcpIDS/HttpIDS$, after having observed a segment violating the TcpStreamNormalization policy, the $TcpIDS$ will be willing to send the alarm $DoesNotRespectTcpStreamNormalization$ and send an a priori undeterministically reassembled message to $HttpIDS$[7]. It's to implement this kind of functionality, that we add the following function to the MIs.

  ```
  val analyze_res_mi_j:state_mi_j -> ds -> ds
  ```

  As it is detailed further, this function is only used by the second, non analyzed, version of the inspection algorithm.

Once we have developed each MI, we can implement the inspect function. The inspection algorithm takes as input a set of different MIs composed with each other. The policy $S1/S2/[Translator1|Translator2]$, can be seen as a tree where S1 is the father of the tree having S2 as root, and the leaves Translator1 and Translator2 as children. To go from the Layer scope, to the Layerings scope, a tree of MI states is used.

```
type state = ... | State_j of state_mi_j | ... (* j in [1,J] *)
(* all the injections from specific state to
   the generic state *)
...
(* injection from state_mi_j to state *)
val to_generic_state_j: state_mi_j -> state
...
(* definition of a polymorphic set
   N.B: for the sake of brevity we don't mention
   the problem of the definition of the comparison function
   over the type 'a, which can be non ordered. *)
type 'a set
(* definition of the state_tree tree *)
type state_tree = Leaf of state | Node of state * (state tree) set
val build_tree: state -> state set -> state_tree

(* we will use this function in the begining of the
   inspect algorithm *)
let get_label_and_children tree =
    match tree with
     | Leaf lf -> lf,emptyset
     | Node (r,c) -> (r,c)

(* map f a, creates the set {f a1,...,f an}
   for ai in a *)
val map: ('a -> 'b) -> ('a set) -> ('b set)

(* project_couple_set {(a1,b1),...,(an,bn)}=({a1,...,an},{b1,...,bn}) *)
val project_couple_set: ('a * 'b) set -> ('a set) * ('b set)
```

[7] This is actually what the stream5 preprocessor of the Snort IDS implements.

```
(* fst (x,y), returns x *)
val fst: 'a * 'b -> 'a

(* snd (x,y), returns y *)
val snd: 'a * 'b -> 'b

(* is_empty e says whether the set e is empty or not *)
val is_empty: 'a set -> bool

(* reduce: computes an element according to a set of elements *)
val reduce: ds set -> ds
(* The definition of the reduce function over ds elements,
   i.e the one that will be used by the inspect algorithm,
   is left open to the developers.
   However this function should verify some conditions we will
   present later so that the inspection algorithm and the
   manipulation of state trees validate some properties.
*)
```

Before detailing the inspection algorithm, we first analyze what happens to a message inspected by the previous rule: $S1/S2/[$
$Translator(input\_language = Lingala, output\_language = Hungarian)/CheckInsult(report\_to = RepublicOfCongoGov)|$
$Translator(input\_language = Hungarian, output\_language = Lingala)/CheckInsult(report\_to = HungaryGov)]$

- Let's say a message from a layering $S1/S'2/Lingala$ ($S'2 \neq S2$) is read by the device. The $S1$ MI inspects the relevant part of the message, accepts it, eventually makes some reassembly, and outputs the $S'2/Lingala$ message. The $S2$ MI then inspects the $S'2$ part of the $S'2/Lingala$ message and send back to $S1$ the decision drop, as it does not allow the messages written in other languages than its own.

- Let's say a message from a layering $S1/S2/HungarianInsult$ arrives. The $S2$ MI then asks both $C1 = Translator(input\_language = Lingala, output\_language = Hungarian)/CheckInsult(report\_to = RepublicOfCongoGov)|$, and $C2 = Translator(input\_language = Hungarian, output\_language = Lingala)/CheckInsult(report\_to = HungaryGov)]$, to analyze $HungarianInsult$. In $C1$, $Translator$ does not recognize $HungarianInsult$ as a part of an $Ingala$ conversation and therefore sends back the $dsc1 = drop$ decision to $S2$. In $C2$, Translator translates $HungarianInsult$, into $LingalaInsult$ and sends it to $CheckInsult$. The $CheckInsult$ then sends back $dsc2 = alarmHungaryGov$ to its father, $Translator$, which forwards it to $S2$. $S2$ gets $dsc2$ from $C1$ and $dsc1$ from $C2$. It then computes $dss2 = reduce(dsc1, dsc2)$ to $S1$. Depending on the definition of the reduce function $dss2$ will be $drop$, $alarmHungaryGov$ or something else. In this case, we do think that $alarmHungaryGov$ would be a good choice, but it's nothing more than a subjective point of view.

We can now define the details of this inspection algorithm:

Listing 2.1: Listing of inspect body

```
(* inspection function of the firewall *)
let rec inspect msg mi_state_tree  =
  let root_state, children_states = get_label_and_tree mi_state_tree in
  match root_state with
    | ...
    | State_j st_mi_j ->
        let msg_layer_j = genmsg_to_msg_layer_<imi[j]> msg in
        let new_state_mi_j,out_j = inspect_mi_j st_mi_j msg_layer_j in
        let new_root_state = to_generic_state_j new_state_mi_i in
        match out_j with
          (* returns the new tree together with the decision taken *)
          | DS_j ds -> (build_tree result_state children_states),ds
          | Out_j out ->
              let generic_out = genmsg_from_msg_layer_<omi[j]> out in
              (* calls recursively the inspect function to
                 all the children_states, and get:
                 - new_children_states: the set of all new children trees
                 - dsset: the set of all the decisions taken
                   by the children trees *)
              let new_children_states,dsset = project_couple_set (
                  map (inspect generic_out) children_states) in
              (* the only remaining thing to do: compute the result *)
              (build_tree new_root_state new_children_states),(reduce dsset)
    | ...
```

One should notice that this first version of the inspection algorithm does not use the *analyze_res_mi_j* functions. We have first not taken this kind of functionality into account for simplicity reasons. Indeed we have not yet formally proved properties of the "inspect" algorithm that uses *analyze_res_mi_j* functions (see Section 2.2.3). This task is left for future work. Both versions of the algorithm have however been implemented .

This algorithm presents interesting properties, indeed it can be classified as being part of *MapReduce* algorithms [110]. *MapReduce* algorithms perform a task over a generic set by means of the *map* operator. Then, a synthesis of all the different answers by means of an ad-hoc *reduce* operator is used as result. As shown by [110], Google has successfully used such methods to efficiently perform important data processing on computer clusters. We have, not during this Ph.D, taken the time to explore the distribution of this inspection algorithm. For example, to inspect messages arriving at 10MB/s, a cluster of computers interconnected at 1GB/s could be helpful with a distributed version of the inspect algorithm. Nevertheless, before executing the non distributed version of the algorithm, it should first be validated.

2.2.2.3 Definition of a Valid Tree

The type checker of *OCaml* validated the type-safety of our algorithm. The inspect function has type

```
val inspect: genmsg -> state_tree -> state_tree * ds
```

The definitions of the projection should however be reminded:

```
let genmsg_to_msg_layer_i gmsg = match gmsg with
    | Msg_i msgi -> msgi
    | _ -> raise ProjectionError
```

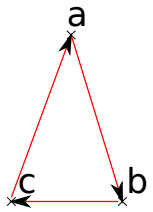


Figure 2.5: Graph example

The case of exceptions is not validated by the static type checking. However with a policy like $.../S1/S2/...$, where the types $msg\_layer\_S_2$ and $msg\_uplayer\_S_1$ are different, all the packets output by $S1$ will produce a *ProjectionError* which can bee seen as a dynamic type error. To statically ensure the absence of such errors, some constraints must be verified on the trees. We remind that $MISET$ is the set of MIs in our midpoint inspection device.

Definition 2 *Let the relation adc, like are_directly_composable, be defined like that:* $\forall(x,y) \in MISET^2, adc(x,y) \Leftrightarrow$ *msg_uplayer_x and msg_layer_y have the same type.*

Among all the packet types, we define two special packet types to validate the root and the leaves of the tree. msg_layer_empty is the type of empty messages, and $msg\_layer\_network$ the type of message that comes from the network to the midpoint inspection device.

Definition 3 *To prevent the presence of leaves that output packets, let us define:*
$\forall x \in MISET, is\_acceptable\_leaf(x) \Leftrightarrow$ *msg_uplayer_x and msg_layer_empty are the same type.*
To validate that the highest inspector of the tree analyzes messages coming from the network, let us define:
$\forall x \in MISET, is\_valid\_root(x) \Leftrightarrow$ *msg_layer_x and msg_layer_network are the same type.*

Checking the validity of a tree is therefore straightforward [59]. For very long protocol layerings a useful feature could be that the interpreter automatically fills missing midpoint inspectors with intermediate MIs.

2.2.2.4 Guessing Missing Parts of Incomplete Policies

Let us take the example of a policy $S_1/../S_i/../S_N$, where all the S_i are default midpoint inspectors. In this case, it would be useful that the midpoint inspection device deduces the valid layering, i.e. that by writing S_1/S_N, the inspector deduces the policy $S_1/../S_i/../S_N$. To check the validity of a layering, checking that each father children S_i/S_j pair are adc is therefore not sufficient, as the absence of intermediary filling layering should be verified too. To do so, other definitions are necessary:

Definition 4 *Let us define direct_sublayers as,* $\forall x \in MISET, direct\_sublayers(x) = \{y \in MISET, adc(y,x)\}$.

Let X be a set of couples, and $Proj(X) = \{x, \exists(x,z) \in X \vee \exists(y,x) \in X\}$, the union of all the elements appearing in the different couples in X. We will define a directed graph, using the usual notation $G = (V,E)$, V being the set of vertexes, and E the set of arcs. For example the graph

$(\{a,b,c\},\{(a,b),(b,c),(c,a)\})$ is depicted in figure 2.5. For a graph $G=(V_G,E_G)$, we will use the following notations, $Edges(G)=E_G$ and $Vertices(G)=V_G$. Finally we will need to formalize the existence of missing parts.

Definition 5 *Let* $(x,y)\in MISET^2$, $y\rightsquigarrow x \Leftrightarrow adc(y,x)\vee\exists z\in MISET, adc(y,z)\wedge z\rightsquigarrow x$.

We can now define the graph we will use to compute the filling parts of a policy.

Definition 6 *Let's define the Sublayers Graph (SG) constructor as for* $x\in S\subseteq MISET, SG(x,S)=(V,E)$,
$E=\{(a,x), a\in direct\_sublayers(x)\cap S\}\cup\bigcup_{a\in direct\_sublayers(x)\cap S} Edges(SG(a,S))$
$V=Proj(E)$

The following property will be useful:

Property 1 $\forall(x,y)\in MISET^2, y\rightsquigarrow x \Leftrightarrow y\in Vertices(SG(x,MISET))$

Let us prove both implications.

- $\forall(x,y)\in MISET^2, y\rightsquigarrow x\Rightarrow y\in Vertices(SG(x),MISET)$:
  This proposition can be proved by recursion on $|MISET|$. Let's take $|MISET|=1$.
  $y\rightsquigarrow x\Rightarrow y=x\in direct\_sublayers(x)$, as x is the only element in $MISET$. Therefore $(x,x)\in Edges(SG(x,MISET))\Rightarrow x\in Vertices(SG(x,MISET))$.
  Let's take $|MISET|=2$.
  $y\rightsquigarrow x\Rightarrow adc(y,x)\vee\exists z\in MISET, adc(y,z)\wedge z\rightsquigarrow x$.
  If $adc(y,x)$ then $x\in Vertices(SG(x,MISET))$.
  Else $adc(y,z)\wedge\neg adc(y,x)\Rightarrow z\neq x$. Therefore $MISET=\{x,z\}$.
  As $z\rightsquigarrow x$ is true $z\rightsquigarrow x$ finishes, which implies $adc(z,x)$.
  If $y=z$, then $\neg adc(y,x)\Rightarrow\neg adc(z,x)$, which contradicts $adc(z,x)$.
  Else $y=x$, $adc(y,z)\Rightarrow adc(x,z)$. With $adc(x,z)\wedge adc(z,x)$, we have $Vertices(SG(x,MISET))=\{x,z\}$ and therefore $x\in Vertices(SG(x),MISET)$.

  Let's assume that for all the $|MISET|=n$ the proposition is true and let's take $|MISET|=n+1$.
  Let $(x,y)\in MISET^2$:

  - If $adc(y,x)$, $y\in direct\_sublayers(x)\Rightarrow(y,x)\in Edges(SG(x,MISET))\Rightarrow y\in Vertices(SG(x,MISET))$.

  - Else $y\rightsquigarrow x\Rightarrow\exists x1\in MISET\backslash\{y\}, adc(y,x1)\wedge x1\rightsquigarrow x$.
    If $x1\rightsquigarrow x$, then $x1\rightsquigarrow x$ is true, and therefore the computation $x1\rightsquigarrow x$ terminated. By exploring the different recursive calls that lead to $x1\rightsquigarrow x$, let us extract the layering $l=x1/.../xn/x$. If y is in l, then we can write $l=x1/../xi/y/x1'.../x'n/x$, and replace $x1$ by $x'1$ in the previous proposition.
    Therefore we can deduce that $\exists x1\in MISET'=MISET\backslash\{y\}, adc(y,x1)\wedge x1\rightsquigarrow x\wedge\forall xj\in l, xj\in MISET'.\ |MISET'|=n$ therefore by the recursion property, we can say that $x1\in Vertices(SG(x,MISET'))$. Moreover by the existence of l, we will have $Edges(SG(x1,MISET))\subseteq Edges(SG(x2,MISET)))\subseteq Edges(SG(x,MISET))$.
    As $adc(y,x1)$ we have $(y,x1)\in Edges(SG(x1,MISET))\subseteq Edges(SG(x,MISET))\Rightarrow y\in Vertices(SG(x))$.

- $y \in Vertices(SG(x,MISET)) \Rightarrow y \rightsquigarrow x$:

  Let's take $|MISET| = 1$.
  $y \in Vertices(SG(x,MISET)) \Rightarrow y = x$ as $Vertices(SG(x,MISET)) \subseteq MISET$. Moreover $x \in Vertices(SG(x,MISET))$ means $Vertices(SG(x,MISET)) \neq \emptyset \Rightarrow direct\_sublayers(x) \neq \emptyset$. Therefore as $direct\_sublayers(x) \subseteq MISET$, $direct\_sublayers(x) = MISET$, $MISET$ being a set of one element, and so $adc(x,x) \Rightarrow x \rightsquigarrow x$.

  Let's take $|MISET| = 2$.
  $y \in Vertices(SG(x,MISET)) \Rightarrow \exists (y,z) \vee (z,y) \in Edges(SG(x,MISET))$.
  $(y,z) \in Edges(SG(x,MISET)) \Rightarrow adc(y,z)$. If $adc(z,x)$ then $y \rightsquigarrow x$. Else, As $|MISET| = 2$, $z = x \vee z = y \vee y = x$. If $z = x$, then $adc(y,z) \Rightarrow adc(y,x) \Rightarrow y \rightsquigarrow x$. If $z = y$, then $\neg adc(z,x) \Rightarrow \neg adc(y,x)$, and $adc(y,z) \Rightarrow adc(y,y)$.
  $\neg adc(y,x) \wedge adc(y,y) \Rightarrow MISET = \{x,y\}$. With $(y,y) \in Edges(SG(x,MISET))$, we have $adc(y,x)$ which contradicts $\neg adc(y,x)$ and therefore $z \neq y$.
  If $y = x$ we have here $\neg adc(z,x) \wedge adc(y,z) \Rightarrow \neg adc(z,x) \wedge adc(x,z)$.

  Let's assume that for all the $|MISET'| = n$ the proposition is true and let's take $|MISET| = n+1$.

  - If $adc(y,x)$ then $y \rightsquigarrow x$.
  - Else $y \in Vertices(SG(x,MISET)) \Rightarrow \exists z \in Vertices(SG(x,MISET))$,
    $(y,z) \in Edges(SG(x,MISET)) \vee (z,y) \in Edges(SG(x,MISET))$.
    * $(y,z) \in Edges(SG(x,MISET))$:

      Let's define $D1 = \{(a,x); a \in direct\_sublayers(x)\}$ and
      $D2 = \bigcup_{a \in direct\_sublayers(x)} Edges(SG(a,MISET))$. As $\neg adc(y,x)$ we have $y \notin D1$. Let's take $D2' = D2 \backslash \bigcup_{(a,b) \in MISET^2} \{(a,y),(y,b)\}$. If $z \in D2'$, then $z \in Vertices(SG(x,MISET'))$ where $MISET' = MISET \backslash \{y\}, |MISET'| = n$. Therefore by the recursion property $z \rightsquigarrow x$, and as $adc(y,z)$, $y \rightsquigarrow x$.
      If $z \notin D2'$, let's take $z' \in D2', adc(y,z')$. z' exists because $y \notin direct\_sublayers(x)$. By applying to z' the same reasoning applied to the previous z we conclude that $y \rightsquigarrow x$.
    * $(z,y) \in Edges(SG(x,MISET))$:
      By renaming $z \leftrightarrow y$ and $y \leftrightarrow z$ we obtain $y \rightsquigarrow x$ together with $z \rightsquigarrow y$ (which is useless for this part of the demonstration).

With this property the problem of missing intermediate layers between S_1/S_N can be solved by checking whether $S_1 \in SG(S_N)$. An algorithm to compute the shortest path from S_1 to S_N gives the shortest filling layering [83]. To have all the possible layerings, one just needs to find all the spanning trees of $SG(S_N)$ with algorithms depicted in [207] and make the union of the paths from S_1 to S_N in all the spanning trees. Last, but not least, the following optimization problem is still open.

2.2.2.5 Optimization of the Inspection Tree

Our inspection algorithm takes a tree as input and scans over all its children to take a decision. We are therefore willing to minimize the nodes of the tree. The question that rises is whether for example the two trees in Figure 2.6 are equivalent ? By equivalent we mean that for all the sequences of

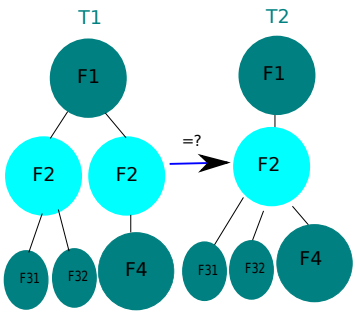


Figure 2.6: Equivalent trees ?

messages both firewalls can observe, they produce the same sequence of decisions. Let's say that after some sequence of messages, the leaves F_{31}, F_{32}, F_4 output ds_{31}, ds_{32}, ds_4. We decide to shorten by r the reduce function in our algorithm. By executing the end of the algorithm, the first tree outputs: $(1) r(\{r(\{ds_{31}, ds_{32}\}), r(\{ds_4\})\})$, and the second tree: $(2) r(\{ds_{31}, ds_{32}, ds_4\})$.

Let's take the example of this r function: $r(\{ds_{31}, ds_{32}\}) = ds_4, r(\{ds_4\}) = ds_4$.
$r(\{ds_{31}, ds_{32}, ds_4\}) = ds_{31}, ds_{31} \neq ds_4$.
Like that, we have:
$(1) r(\{r(\{ds_{31}, ds_{32}\}), r(\{ds_4\})\}) = ds_4$,
$(2) r(\{ds_{31}, ds_{32}, ds_4\}) = ds_{31} \implies (1) \neq (2)$
And therefore, the two trees are not equivalent. We use $\wp$ to express the set of the subsets of a set, i.e. $\wp(\{1,2\}) = \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$, and define the following:

Definition 7 *Let* **DS** *be a set of decisions containing the following properties: reduce* : $\wp(\mathbf{DS}) \rightarrow \mathbf{DS}$, $\forall X \in \wp(\wp(\mathbf{DS})), reduce(\{reduce(x), x \in X\}) = reduce(\bigcup_{x \in X} x)$

We use analogy with election algorithms to illustrate this property. In indirect elections, people first vote for special electors, who, in a second round, make the final choice. By taking as X, the set of people electing their special electors, the election of super electors can be seen as $\{reduce(x), x \in X\}$, and the choice of super electors as $reduce(\{reduce(x), x \in X\})$. In direct election, all the people vote at once which can be seen as $reduce(\bigcup_{x \in X} x)$. The property of our reduce function can be seen as a constraint to prevent situations similar to United States of America's 2000 presidential election, an indirect election system. Indeed in this case, the elected president would have lost the election in a direct system, and vice-versa.

Let **IT** be the set of valid inspection trees. A valid tree is said to be *Reduced* if all its paths from the root to the leaves produce a unique result given a specific sequence of messages. We consider that two valid trees are equivalent if they produce the same sequence of decisions for all the sequences of input packets. Finally, for a a valid tree x, we define as $paths(x)$, the set of paths from the root to all of its leaves. Based on those definitions the analysis of our algorithm proved the Theorem **??**. Let

$x \in \mathbf{IT}, x\ Reduced, \forall y \in \mathbf{IT}, equiv(x,y) \Leftrightarrow paths(x) = paths(y)$. From this theorem we can deduce that the minimum equivalent tree of a reduced tree is a factorized tree, where there are no duplicated fathers. The algorithm in Listing 2.3 computes such a tree by using the primitives we added in 2.2.

Listing 2.2: Listing of join primitives

```
(* union e f, creates the set e U f *)
val union: ('a set -> 'a set) -> ('a set)

(* choose e, chooses an element over a set e, and rises
an exception if the set is empty *)
val choose: ('a set) -> 'a

(* diff e f, computes the set e \ f *)
val diff: ('a set) -> ('a set) -> ('a set)
```

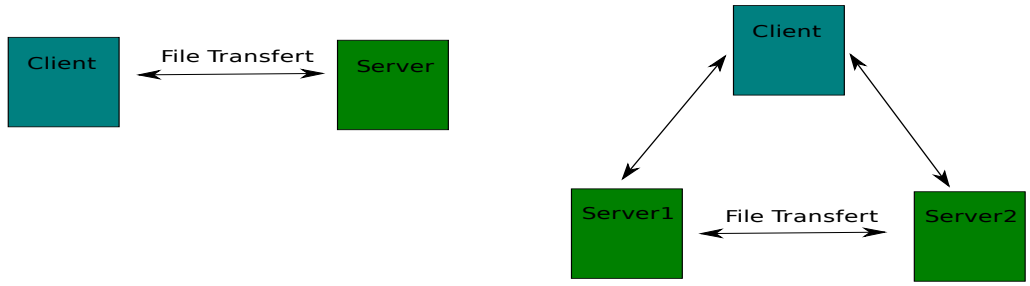
Listing 2.3: Listing of tree factorization

```
let label tree = fst (get_label_and_children tree)

let rec rjoin trees =
  if is_empty trees then emptyset
  else
    let l = map label trees in
    let rec group_same_label remaining res x  =
      if is_empty remaining then build_tree x (rjoin res)
      else
        let elm = choose remaining in
        let continue = diff remaining (singleton elm) in
          if (label elm) = x then
           let nres = union res (children elm) in
            group_same_label continue  x
          else group_same_label continue res x
    in
      map (group_same_label trees emptyset) l

let join (tree: mi tree) = build_tree (label tree) (rjoin (children tree))
```

We have in this section described the model we use to analyze the first version of the inspection algorithm. By this model we provide algorithms to compute:

- the validity of inspection trees,
- the missing default layers of an incomplete layering,
- the optimized version of valid inspection trees.

The first version of the inspection algorithm is based on a static tree configured by the policy a user wants to implement. However, dynamically negotiated layerings, as for example FTP data channels, creates the need for modifying dynamically the inspection tree. The following section describes how the second version of the inspect algorithm permits to dynamically modify a given inspection tree.



(a) First usage (b) Second usage

Figure 2.7: The two FTP usages

2.2.3 Handling of Dynamically Negotiated Layering

We refer as dynamically negotiated layerings, the layerings that depend on parameters negotiated during a previously instantiated layering. Inspecting the *File Transfer Protocol* (FTP) [36] protocol inspection permits to evaluate whether dynamically negotiated layerings are handled by one midpoint inspection device. The FTP protocol is used to transfer files among computers, organized in two different ways as depicted in Figure 2.7:

- in 2.7a, Client and Server negotiate a transfer from Client to Server, or from Server to Client,
- in 2.7b, Client negotiates a transfer from Server1 to Server2 or from Server2 to Server1.

To do so, FTP uses two different communication channels. The first communication channel negotiates the parameters to do the file transfer, called *control channel*. The second communication channels is dynamically computed using the parameters exchanged in the *control channel*. This dynamically computed *data channel* is used to transfer the file. The layering used by the *control channel* is $IPv4/TCP/FTPControl$. As said earlier, this layering negotiates a set of parameters $p < MI >$ to configure the *data channel*, and its layering: $IPv4(pIPv4)/TCP(pTCP)/FtpData(pFtpData)$. In most of nowadays endhost operating systems, the new layering is computed by means of the *socket library*. Indeed, some primitives of the library[8] gives the possibility to one user to configure the creation of a predefined set of monolithic protocol layerings. However our inspection algorithm works using a different paradigm and needs therefore to find another solution to this problem. Therefore in our case, the $FTPControl$ MI needs to find a way to dynamically add a temporary $IPv4(pIPV4)/TCP(pTCP)/FTPData(pFtpData)$ into the inspection tree. This problem implies three issues:

1. how to compute the different set of parameters $p < MI >$,

[8] bind,listen and connect for instance.

2. how to compute the layering,

3. how to integrate it to the inspection tree.

The difficulty of computing $p < MI >$ resides in the fact that according to the messages it sees, the $FTPControl$ inspector may not know some values to compute $pIPv4$, or $pTCP$. Moreover, the user configuring the policy of the firewall does not a-priori know the values in $\{pIPv4, pTCP, pFTPData\}$ that are computed during the observation of messages in $FTPControl$.

We chose to police these dynamic layerings with variables whose values have to be set before being integrated to the inspection tree. For example in the FTP protocol, the messages exchanged in the *control channel* permits to configure parts of the TCP tuple, i.e $client\_address, server\_address, client\_port, server\_port$ of the *data channel*. Therefore we define the variables $related\_client$ and $related\_server$, that represent the dynamically negotiated parameters of the TCP tuple in the *control channel*. Moreover, to give the ability to express the link between the *control channel* and the *data channel*, we define the variables *previous_client* and *previous_server* that represents the addresses or ports , in function of the argument they configure, of the TCP tuple of the control channel computing the dynamical layering.

These variables are dynamically substituted by means of the of the analyze_res_mi_j functions. Indeed, the second version of the inspection algorithm has the modifications presented in the Listing 2.4.

Listing 2.4: Listing of the modification of the inspect function's body

```
| Out_i out ->
  ...
 (* takes the set of all new decisions *)
 let dsset = map get_second_element
                 children_results in
 let dsout = analyze_res_mi_j new_root_state
                              (reduce dsset) in
 (build_tree new_root_state new_children_states,
                              dsout)
```

Once all the variables of a layering are substituted the layering is computed. Finally, to integrate the layering into the inspection tree, we add the decision *add_child* taking as parameter a dynamically configured layering into the decision set *DS*.

Let us illustrate more in detail how this mechanism works. Two different messages analyzed by $FTPControl$ configure the different p values, the *PASV* and *PORT* messages. Without entering into the details of such messages, the Figure 2.8 shows how the computation is done. The negotiated parameters influence the values of the new layering depicted on the right part of the Figure 2.8. These parameters encode the source and destination $IPv4$ addresses, together with the source and destination ports in TCP. As said earlier, we identify as $related\_client$ and $related\_server$ the values of addresses of ports negotiated by the *PASV* and *PORT* messages seen by the $FTPControl$ MI. The $previous\_client$ and $previous\_server$ identify the values of addresses and ports of the control channel layering. The arguments to configure the $FTPControl$ MI has therefore an optional argument $data\_channel$, to police the expected dynamic layerings. Let's say the user configures the following rule: $FTPControl(data\_channel = IPv4(src = previous\_server, dst = previous\_client)$
$/TCP(sport = 20, dport = related\_server)/FTPData)$.

After observing the *PASV* message, $FTPControl$ computes the value of $related\_server$ and outputs the decision add_child $child$, with $child = IPv4(src = previous\_server, dst =$

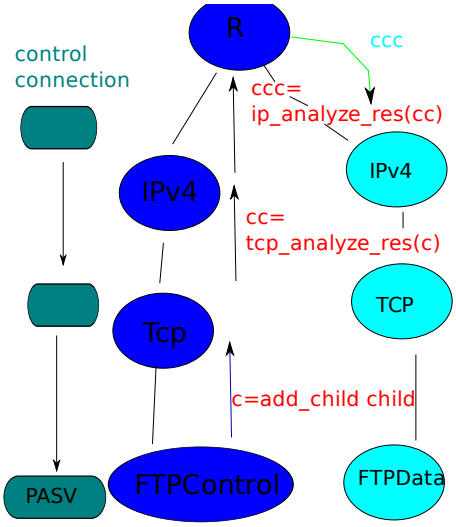


Figure 2.8: Add child

$previous\_client)/TCP(sport = previous\_client, dport = 1026)/FTPData$. Then TCP analyzes the result, and updates child, producing for example $cc = IPv4(src = previous\_server, dst = previous\_client)/TCP(sport = 20, dport = 1026)/FTPData$, and similarly $IPv4$ outputs: $ccc = IPv4(src = 0.0.0.1, dst = 0.0.0.2)/TCP(sport = 20, dport = 1026)/FTPData$. The root of the tree then appends this new layering to the set of temporary layerings and inspects the messages according to the updated tree. This version of the inspection algorithm is not validated by the Theorem **??**. Indeed the situation is more complicated here. If we go back to the examples we mentioned to illustrate the equivalence of factorized tree in the Figure 2.6 and call ar the function $analyze\_res\_F_2\ st_{F2}$, the equations become: $(1)r(\{ar(r(\{ds_{31}, ds_{32}\})), ar(r(\{ds_4\}))\})$
$(2)ar(r(\{ds_{31}, ds_{32}, ds_4\}))$. The necessary conditions on the ar and r functions to maintain the equivalence are left for future work.

We have shown in this section the modification done to the inspection algorithm to handle dynamically negotiated layerings. In the following section we describe the implementation of this version of the inspection algorithm.

2.3 Implementation and Benchmarking of the Midpoint Inspection Device

This section presents Luth, the midpoint inspection device prototype we develop. This prototype is stressed on two different cases studies involving firewall functionalities. The first one addresses the DNS protocol, the second one FTP.

Figure 2.9: Luth's logo

2.3.1 Name and Targeted Platform

Luth is a new kind of midpoint inspection device that can be configured with great flexibility. Among others, its configuration language can be used to express detailed inspections, as in policies with long layerings or shorter ones, if for example the user does not want the device to perform heavy computing. Inspired by the analogy made by computer firewalls, Luth stands for Lur Ur Ta Haize, which means in Basque language, Earth, Water And Wind. Indeed, fire can be stopped by water and earth or propagated by wind. The same way Luth can use different ways to inspect Internet communications, or to let some anomalies bypass itself. This is the reason of Luth's triskel hat. The triskel is a celtic symbol representing the four elements: fire, water, wind and earth standing in the middle of the symbol. That is why the triskel hat has only two branches: the fire is gone (Luth does not fight fire with fire). Finally Luths are the biggest turtles overseas. Indeed, when most of other inspection devices are mainly concerned about processing speed, Luth focuses on the expressiveness and the correction of its algorithms and implementations. Indeed, quoting Aesop's *The Tortoise and the Hare*, "Slow and steady wins the race."

2.3.2 Different Case Studies

The implementation of Luth is written in *OCaml*, making a quite extensive use of the Melange library to safely and efficiently parse packets [169].

To perform online inspections, the binding of the libnetfliter_queue [26] to OCaml has been developed with the help of the stub code generator CamlIDL [8]. This library offers some system-calls to communicate with Netfilter, GNU/Linux's framework to implement firewalls and NAT routers. The packets to inspect are selected by means of the *iptables* utility [26]. Let us illustrate how this library works with the following example:

```
iptables -s 192.168.1.254 -d 192.168.1.1 -A FORWARD -j NFQUEUE\
--queue-num 0 #R1
iptables -d 192.168.1.1 -d 192.168.1.254 -A FORWARD -j NFQUEUE\
--queue-num 1 #R2
```

R1 tells Netfilter to send all the packets forwarded by the operating system having the *IPv*4 source address 192.168.1.254 and the 192.168.1.1 destination address to a program in userspace polling the queue identified by the number "0". The program in question informs Netfilter that he will be polling one queue by means of the function provided by the libnefilter_queue library. R2 tells the packets from 192.168.1.1, to 192.168.2.254 to go to the software polling the queue "1". The same program can poll multiple queues by means of the "select" system-call. After having inspected the packet, the decision is communicated to Netfilter using other functions of the library. The reader should notice that the decisions

Netfilter offers to give are different from the decisions Luth can take. Therefore Luth translates to its best the *ds* decisions to Netfilter decisions. We won't go further for now, as more details will be given in Section 4.2.2.

To perform offline inspections, a binding of the libpcap library [45] to OCaml, mlpcap [23], has been modified to remove identified bugs.

To check the correction and measure the performance of the current implementations we first check it under two different experiments involving filtering decisions like firewalls do. The first experiment targets DNS tunneling and the second, the FTP protocol.

2.3.2.1 DNS tunneling

To get the Internet for free in commercial hotspots, a DNS tunneling technique can be used quite efficiently. Indeed to redirect new clients' first web page requests to a portal where the payment system is explained, this kind of hotspot needs to let the client make DNS[9] requests. However, whereas this service only needs to make hostname resolutions, the hotspot usually let the users ask all types of DNS requests. Users trying to get Internet for free use the semantically vast enough TXT[10] requests, to encode TCP segment into DNS messages [136, 187]. To sum up, those bypassing layerings can be seen as IPv4/UDP/DNS(messages=TXT)/Tunnel/...

To stress our firewall in this situation we choose to install the *dnsmasq*[11] *DNS* forwarder in the server. *dnsmasq* will forward all the *DNS* requests it receives to a real *DNS* server. We then install the *apache*2[12] web server on *S*, and generate 20 html pages. Each of these pages have 10 images. Those images size follows a 1.2 pareto distribution [109]. We multiply the pareto coefficients by 10240 to obtain image sizes from 10 to 500KB. The simulation software in the legitimate client makes a host name resolution request to a predefined set of domain names, and then downloads one of the 10 web pages only if the request was successful.

To implement the hotspot thief, we register a domain name to make the *DNS* tunneling possible. We choose the *dns2tcp* [136] tunneling framework and install the client software in the client with the address 192.168.2.10 in Figure 2.10. Then we simulate a client that periodically uses the *SSH* protocol over this *DNS* tunnel. After getting a shell on the distant server, we ask the server to display the contents of one of the 100 files generated with the same pareto distribution using the *cat* utility. With a coefficient of 10240, we obtain file sizes from 10 to 189 KB. We associate a timeout of 10s to the completion of these requests. To demonstrate that Luth can handle cyclic layerings we implement the filtering of this scenario into self-encapsulated GRE tunnels as depicted in Figure 2.10[13]. All the different hosts are present in *laasnetexp*, a testbed environment in our laboratory [193]. The computers used in this experiment present the characteristics described in the Figure 2.11.

The server and the firewall, i.e the server executing Luth, are connected by the 192.168.1.0/24 network. To reach the 192.168.10.0/24 network, the firewall needs to go through *n Generic Routing Encapsulation* (GRE) [40] tunnels. The server, has to go through the same tunnels to reach the 192.168.2.0/24 network. Finally both clients use the 192.168.2.254 gateway to reach the 192.168.10.0/24 network. The implementation of the *n* encapsulation is achieved using the *iproute* package of the Debian project. The

[9] `http://www.ietf.org/rfc/rfc1034.txt`

[10] `http://www.isi.edu/in-notes/rfc1464.txt`

[11] `http://thekelleys.org.uk/dnsmasq/doc.html`

[12] `http://httpd.apache.org/`

[13] The reader should notice that these tunnels are not necessary for both clients to perform their downloads. Indeed, the tunnels only serve to illustrate the expressiveness of Luth.

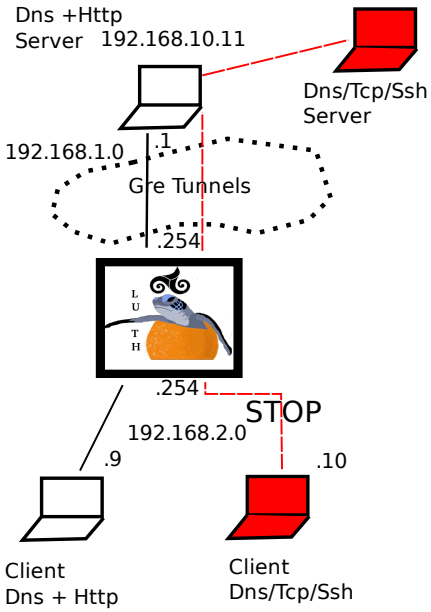


Figure 2.10: Tunneled DNS filtering. The 192.168.2.9 client downloads files in the HTTP server in 192.168.10.11 by means of DNS and HTTP protocols. The malicious client in 192.168.2.10 uses a hidden channel by means of the DNS protocol to download files in the SSH server.

Network interface cards (NIC)	Gigabit Ethernet controllers: Broadcom 5721J and IntelPro1000PT
Interconnection	Gigabit Ethernet: Cisco Catalyst 6504
Processors	Dual Core Xeon 3050
Memory	2GB of 667MHz Dual Rank ECC Memory (2x1GB)
Operating System	Debian GNU/Linux (2.6.18 kernel)

Figure 2.11: Computer and interconnection network's characteristics.

developed MI together with their arguments are depicted in the manual depicted in Annex 3. We configure 20 tunnels between the firewall and the server. We chose this value because approaching 35 tunnels, both tunnel extremities have kernel panics. This is surely due to a segmentation fault in the kernel code to handle the routing and the encapsulation process in the Linux kernel. We have not investigated this issue for a lack of time. 20 is therefore a value representative enough that does not involve the crashing of operating systems during the experiments.

To stress our firewall, we first randomly generate a scenario. 5 clients download in each hosts, random files during one minute every t, where t follows a law having an exponential distribution of mean 0.01ms. We measure for each downloads, the time from the start of the client application to the end of it divided by the size of the downloaded file, what is noted bandwidth. For a failed download the client returns -1 as download bandwidth.

We run the experiments in three cases studies.

- First, no firewall is put in the midpoint inspection device, labeled Ref.
- Second, a userspace firewall we have written for benchmarking purposes in C, labeled Cfw. This firewall accepts all the packets after having read them by means of of the libnetfilter_queue library.
- Third, with Luth configured with a certain rule in it, labeled $Luth$.

We configure Luth using the rule depicted in Listing 2.5.

Listing 2.5: DNS rule

```
IF(name=idx2,addrs=default,messy=yes,lo=192.168.1.254);;
INSP(L2N/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
    Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
    Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/[ Dns | Http ]);;
```

The filtering tree computed by the firewall which fills the holes between *Dns Http* and *Gre MIs* is depicted in Listing 2.6.

Listing 2.6: Dns rule's MI Tree

```
L2N/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
    Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
    Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Ttl(min_ttl=10)/[
    Tcp/[Port(sr_port=53)/Dns | Port(sr_port=80)/Http] |
    Udp/Port(sr_port=53)/Dns]
```

The Figure 2.12 shows the bandwidth of the first of the five unauthorized clients without Luth (other clients present similar results). With Luth all malicious downloads output -1, showing all such requests have successfully been dropped. The five legitimate clients perform about 2000 downloads during the experiment. Their download bandwith reaches $10^6 B/s$, going from 1 up to $10^7 B/s$ for the fastest downloads during the experiment without midpoint inspectors. Figure 2.13 shows the frequency of the downloads in the different slots from 0 to 9.3 MB/s. We see that the curve labeled Luth seems to be the translation to the left of both of other curves, meaning that the bandwidths obtained with Luth are overall slower than in both of the other cases. We'd like here to quote Donald Knuth that gave the following advice: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only

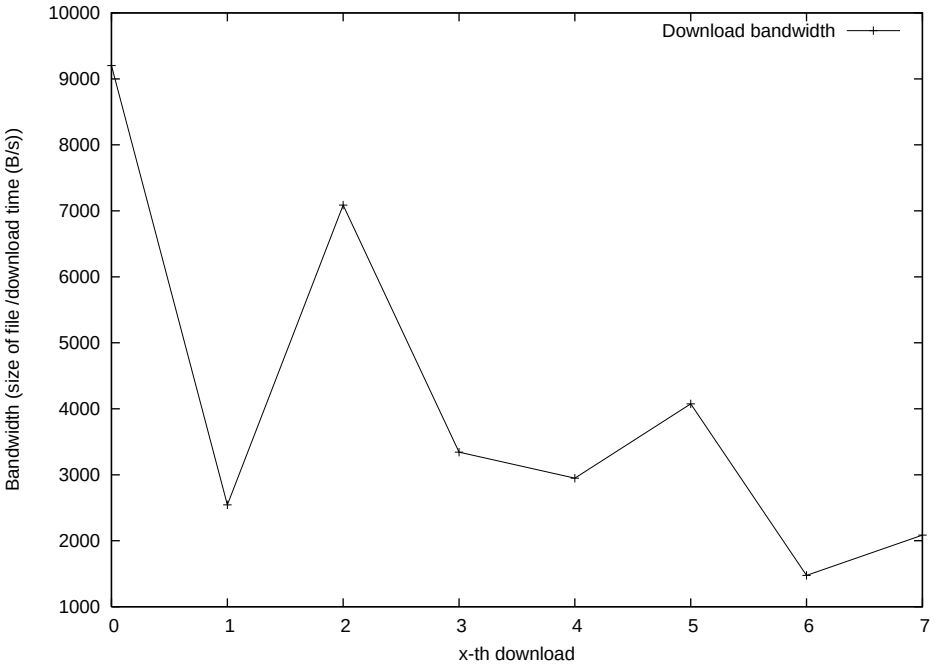


Figure 2.12: Performance of malicious DNS clients in the unfiltered case.

after that code has been identified" [155]. Our prototype is mainly concerned about expressiveness and correction, and not performances. That's why we have not during this Ph.D taken the time to identify these critical sections.

By comparing Luth's bandwidth curve, to Cfw and Ref curve, both the side-effects due to being in userspace and due to the performed inspections are taken into account. However, we want to just evaluate the side-effects due to the performed inspections. To have a better understanding of the difference among these three distributions, Figure 2.14 plots the following distributions representing the 2000 legitimate client downloads times:

- Ref - Ref': Ref' being the experiment Ref executed another time.
- Cfw - Cfw': Cfw' being the experiment Cfw executed another time.
- Ref - Cfw.

We see in Figure 2.14 that the difference of Ref - Ref' is negligible: the network state, and respective client and server application states are stable enough during two experiments. The negative results obtained in Ref - Cfw shows that some downloads have been faster in the C firewall than in the Ref experiment. One explanation of such a trend is that by slowing down some downloads, the following downloads reach the HTTP server faster and are therefore served more quickly than in the Ref experiment. Even if interesting, we have not investigated this phenomenon more deeply. What should be noticed here is that Cfw - Cfw' looks like Ref - Cfw. By this, we see that the scheduling, queueing, and context-switching, induced by the userspace firewall creates a non negligible variability, independent of the inspections done by the firewall (else Cfw - Cfw' will look like Ref - Ref').

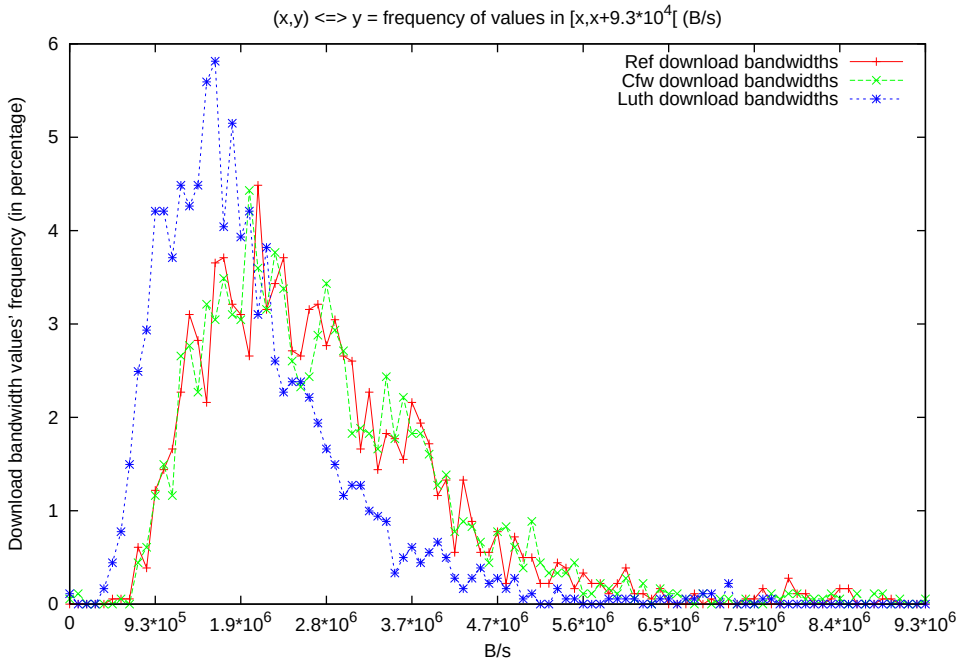


Figure 2.13: Distribution of downloads bandwidth obtained with dns filtering using default configuration. The download bandwidths reached in Ref and Cfw experiments are mostly in the interval $[1.9*10^6, 2.8*10^6[$ B/s, when the ones obtained during Luth experiment are in $[9.3*10^5, 1.9*10^6[$.

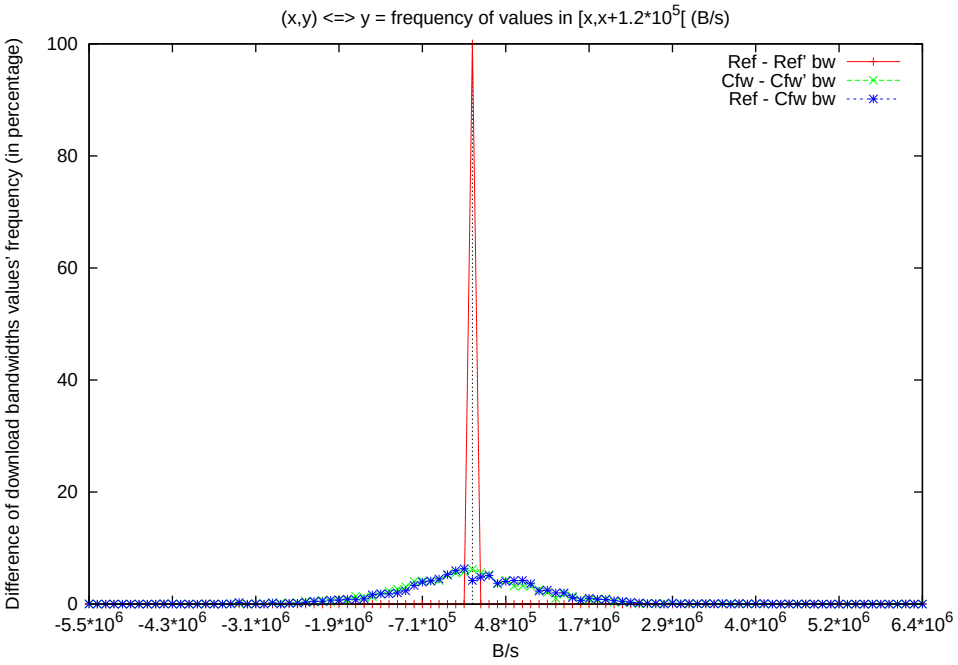


Figure 2.14: Evaluation of the side effect provoked by a userspace firewall. Being in userspace creates a variability that influences the bandwidth results as much as the performed inspection ($(Cfw - Cfw' \simeq Ref - Cfw) \neq Ref - Ref'$).

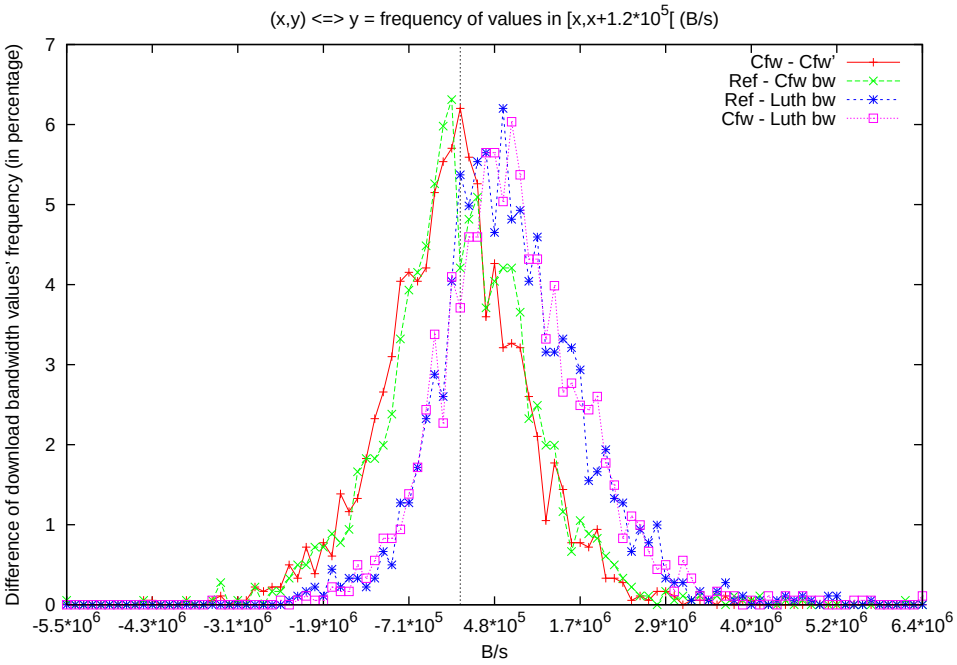


Figure 2.15: Distribution of download bandwidth differences in DNS experiment with default configuration.

To isolate userspace side-effects and measure the ones due to Luth inspections, we compare the difference between Luth Cfw and Luth Ref, with the differences between Cfw and Cfw', or Ref and Cfw, the ones exhibiting the side-effects due to being in userspace. Therefore, Figure 2.15 plots the following distributions:

- Cfw - Cfw',
- Ref - Cfw,
- Ref - Luth,
- Cfw - Luth.

As depicted in Figure 2.15 the curves Cfw - Luth, and Ref - Luth have similar shapes. The same way the curves Cfw - Cfw', and Ref - Cfw present similar shapes, that seam to be a translation from about 1MB/s to the left. This translation shows a non negligible slow down induced by to the inspections performed by Luth.

The rule in Listing 2.7 aims at specifying a new policy to the situations. If the computations done by the hotspot's HTTP server are trusted[14], inspecting the layering involved in the HTTP session (TCP/HTTP) can be considered as useless. In this case, we can configure Luth to accept all TCP packets without performing any inspection on them, and concentrate the work on DNS filtering with the rule described in Listing 2.7.

[14]Indeed in this peculiar case, the DNS tunneling problem comes from the DNS server.

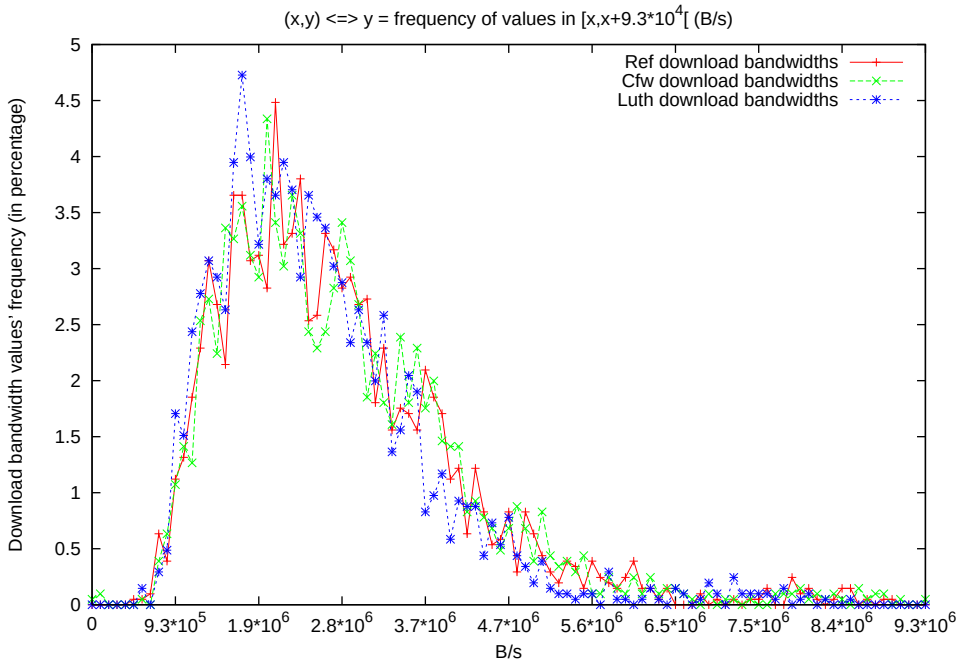


Figure 2.16: Distribution of downloads bandwidth obtained with DNS filtering using optimized configuration.

Listing 2.7: Dns custom rule

```
IF(name=idx2,addrs=default,messy=yes,lo=192.168.1.254);;
INSP(L2N/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
    Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/
    Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/Gre/Ipv4/[Tcp(tcp_only=yes)|Udp/Dns]);;
```

The figures 2.16 and 2.17 show that with this configuration, the slow down induced by Luth's inspection is less visible.

In this case study, we have shown that Luth could be used to efficiently implement a policy that drops DNS tunneling in open hot spots.

2.3.2.2 FTP filtering

We evaluate in this section how Luth handles dynamically negotiated protocol layerings, by dropping passive FTP clients while accepting active ones. We describe the results given by the experiment depicted in Figure 2.18 with N=1 client. We generate 100 files of random size using files following a pareto distribution of coefficient 1.2. After multiplying the coefficients given by the generator by 1024*100, we obtain 100 files whose size vary from 100KB to 6MB. We choose *vsftpd* [15] to implement the FTP service. We grant anonymous access to our clients. We implement their downloads with the *wget*[16]

[15] http://vsftpd.beasts.org/

[16] http://www.gnu.org/software/wget/

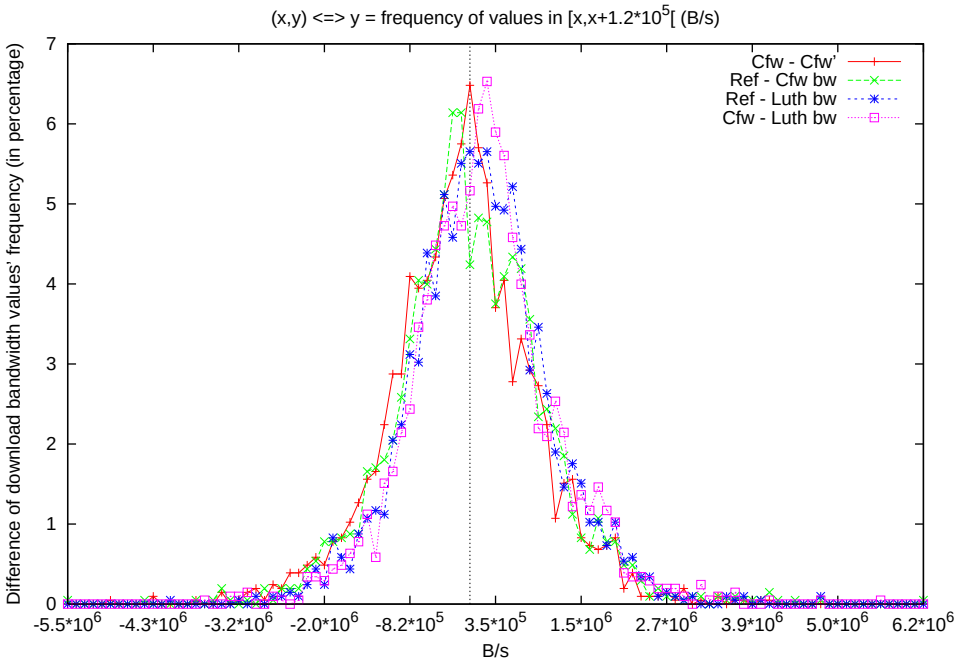


Figure 2.17: Distribution of download bandwidth differences in DNS experiment with optimized configuration.

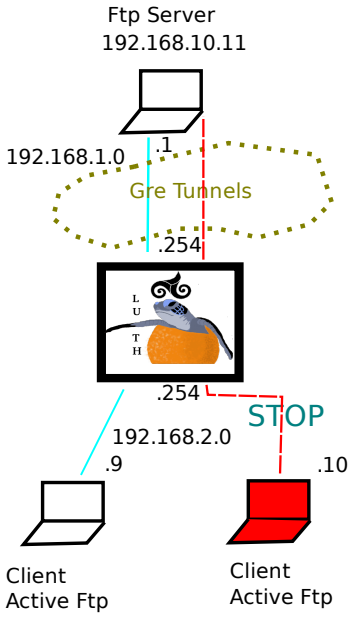


Figure 2.18: Tunneled ftp filtering.

Figure 2.19: Performance of the unauthorized active ftp client without filtering.

utility. We configure *wget* to use active *FTP*, to try twice before considering a download has failed, and set a timeout of 10 s[17]. Here again, an error returns the -1 bandwidth.

The policy we implement is that the client with IP address "192.168.2.9" is allowed to do active FTP downloads, and the client with the address "192.168.2.10" passive downloads. The downloads with the address"192.168.2.10" must therefore all be dropped.

The rule described in Listing 2.8 specifies such a policy.

Listing 2.8: Ftp rule

```
IF(name=idx2,addrs=default,messy=yes,lo=192.168.1.254);;
INSP(L2N/<encapsulations>/Tcp/[
    TPort(cl_addr=192.168.2.9,sr_addr=192.168.10.11,sr_port=21)/
    Ftp(ftpt=active,data_ch=L2N/<encapsulations>/Tcp/
    TPort(cl_addr=previous_server,sr_addr=previous_client,cl_port=20,
    sr_port=related_server)/CSend)|
    TPort(cl_addr=192.168.2.10,sr_addr=192.168.10.11,sr_port=21)/
    Ftp(ftpt=passive,data_ch=L2N/<encapsulations>/Tcp/
    TPort(cl_addr=previous_client,sr_addr=previous_server,
    cl_port=1024::65535,sr_port=related_server)/CReceive)]);;
```

The Figure 2.19 show the results obtained by the active client in .10 host, without Luth. With Luth, all the unauthorized downloads report a -1 value, showing that the policy is correctly implemented. The figures 2.20 and 2.21 show the performances of the filtering device.

[17] wget -t 2 -T 10 –no-passive-ftp <url> -O /dev/null

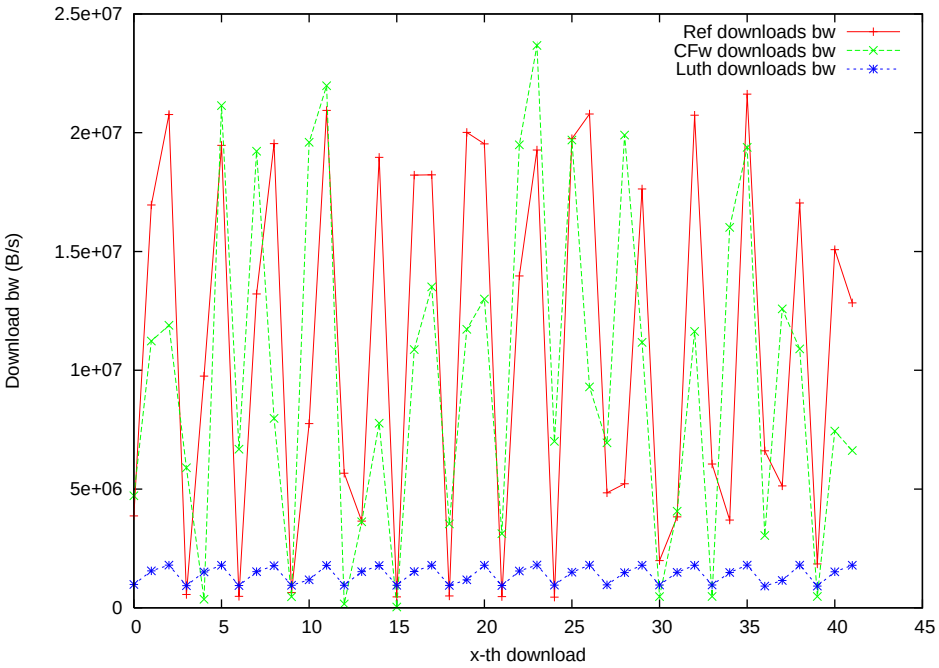


Figure 2.20: Performance of ftp filtering.

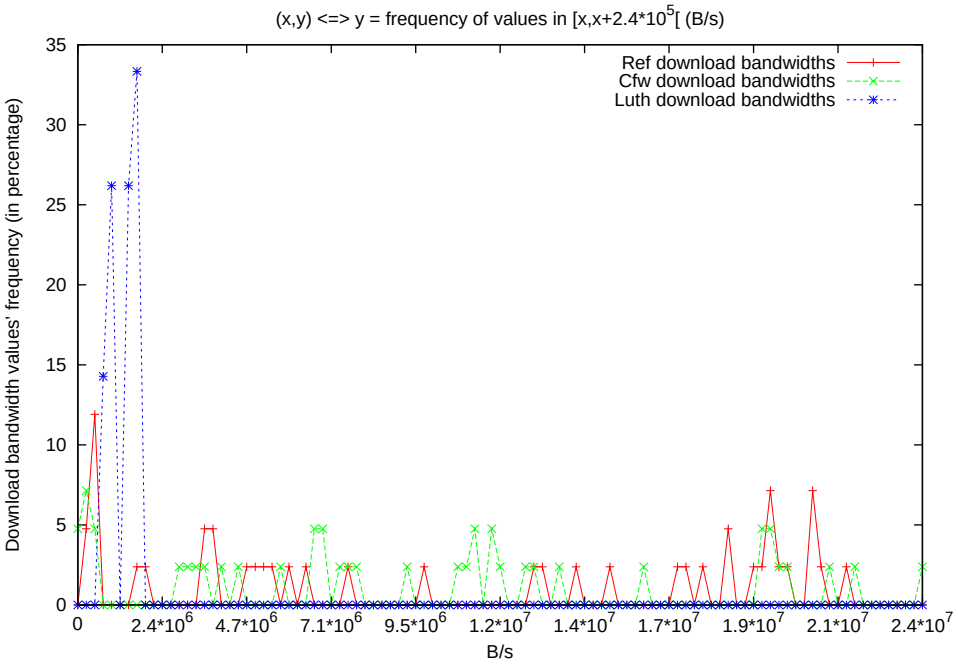


Figure 2.21: Distribution of downloads bandwidth obtained with ftp filtering.

We see that the handling of the creation of dynamic connection works, but needs to be optimized. Indeed the downloads with Luth reach around 1MB/sec when the FTP downloads reach 20MB/s. To reach this magnitude of speed, the migration of filtering computations to kernel space, and parallelization seams mandatory, as the filtering tree grows with each new data channel. These problematics are left for future work.

2.4 Conclusion of Chapter 2

We have presented in this chapter the tool we use to monitor malicious communications. The expressiveness of this midpoint inspection device enables to monitor up to our knowledge all Internet protocol layerings, while providing correction and optimality checking. The DNS and FTP case studies show that Luth, the userspace prototype of our midpoint inspection device correctly implements policies involving 20 iterations of a cyclic layering. Up to our knowledge other existing midpoint inspectors cannot handle such policies.

There are however still some main and unresolved issues:

- The validation of the presented demonstrations by a proof-assistant.
- The analysis of the second version of the algorithm (the one with analyze_res functions). Among others, in its current state, the safety of dynamically computed layerings is not statically guaranteed. The way web interaction have been typed in [67] or the great work done in [89] can provide a way to statically verify the validity of dynamically computed layerings. Moreover, this last work provides a solution to implement the inspect function without creating the universal type of states and packets, and without using Generalized Algebraic Data Types (GADT) that are still not implemented in for example the OCaml language. These two works are therefore worth studying as Sebastien Mondet suggested it to me.
- The parallelization of the inspection algorithm.
- The demonstration of the correction of the overall implementation.
- The optimization of performances.

After having presented its performances on two different experiments, we are now ready to show two real case studies. The first study addresses the monitoring of malware downloading honeypots. This study is interesting because the policy to monitor these honeypots do not follow the classical: drop potentially malicious packets, accept all the others, paradigm. Indeed, malicious packets should be accepted so that the intrusion attempts are emulated by the malware downloading honeypot. However, the download requests the honeypot generates can in fact be intrusion attempts towards third-party ADPS. Therefore, such malicious download requests must be dropped. The Chapter 3 shows how Luth correctly implements this original policy.

3
Monitoring Malware Collection

> "The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.)". Ken Thompson.

This chapter first explains why malware collection should be monitored. To do so, we show how malware downloading honeypots can be used as code injection attack proxies. The presented attacks make the honeypots generate download requests, which are in fact code injection attacks. Indeed we successfully inject and make specially developed vulnerable servers execute two different type of injected codes:

- x86 code, what is usually called shellcode,
- SQL code, involved in SQL injection attacks.

By these attacks, we motivate the need for monitoring malware downloading honeypots. The monitoring of a malware downloading honeypot implies first, to let some malicious traffic go to the honeypot, so that intrusion attempts are emulated. Then, the eventual malicious traffic towards third party ADPS should be dropped. Said otherwise we need to implement quite an original policy:

- be paranoid enough to drop the sequences of packets that can lead to attacks on a third party Automatic Data Processing System (ADPS),
- be permissive enough to accept the sequences of packets that make the honeypot emulate intrusions and download malware.

The second section deals with this problem. We first describe the developed Midpoint Inspector (MI) by taking a close look to the monitoring of TCP sessions' establishment. Then we evaluate the actual number of dropped and accepted packets together with the performances of the inspection. This evaluation is done by means of an offline analysis on traffic dumps taken to record unmonitored honeypots' sequence of packets.

3.1 The Need for Monitoring Malware Collectors

This section aims at showing both the SQL an x86 code injection attacks presented in the introduction. First, the honeypot software involved in the experiments are described: Nepenthes and PHP.HoP. Then we show how we make:

- Nepenthes perform an SQL injection attack.

- PHP.HoP perform an shellcode injection attack.

Among the different honeypots we presented in 1.2.3, we have chosen the Nepenthes platform [64] to download malware. This honeypot emulates a set of vulnerabilities to extract the parameters sent by an intruder to make the attacked computer download and execute the malware of its choice. The main advantage of the Nepenthes honeypot is that it just stores the binary and does not execute it. Indeed, the unsupervised execution of malware can probably lead to attacks against a third-party ADPS. Therefore, by just storing the binary, the possible intrusions perpetrated by the emulation of the attack are restricted. Trusting endhost software is what conducted to the apparition of distributed malicious activities in the Internet. Therefore, we have been willing to check if downloading malware by means of a honeypot, an endhost software, could conduct to a propagation of an attack. We performed a similar analysis on the PHP.HoP [191] web-worm[1] downloading honeypot too. Instead of auditing the emulation codes for possible vulnerabilities, we audited how the download requests were computed to check if a third-party could be attacked like that. After some investigations, we found a way to conduct two kind of intrusions on customized servers we developed for this purpose: an SQL injection and a shellcode injection attack. The way download requests are computed is more restricted in the PHP.HoP honeypot. Therefore we only managed to send a shellcode injection with it. We managed to launch both attacks using the Nepenthes honeypot. We present a summarized version of the attacks in the following sections. For more details the reader is invited to read the technical report depicted in [139].

3.1.1 Making Nepenthes Launch an SQL Injection Attack

This section describes how we make Nepenthes generates an SQL injection attack that changes the password of a user in a web application. The web application uses a database implemented by the MySQL [24] *Relational Database Management System* (RDBMS). The table is created with the Listing 3.1.

Listing 3.1: Web application table

```
use forSqlInjection;

CREATE TABLE users (
        username CHAR(10),
        userpassword CHAR(10),
        userlastname CHAR(10)
        );

INSERT INTO users (username,userpassword,userlastname)
VALUES ('admin','aj6che:$','');
```

The web application is written in PHP [30], one of the WebL we mentioned before (Section 1.1.1.3).

Listing 3.2: Web application code

```
/* ... */
$userName=$_GET["name"];
$oldPassword=$_GET["oldPass"];
$newPassword=$_GET["newPass"];
```

[1] A specific kind of malware that exploit web applications vulnerabilities.

```
$sqlrequest = "update users set userpassword ='"
              .$newPassword."' where username='"
              .$userName."' and userpassword='"
              .$oldPassword."'";

$request = mysql_query($sqlrequest)
           or die ('Invalid sql request: '.$sqlrequest);
if (mysql_affected_rows($link) == 0) {
   echo 'Invalid user password or name';
}
else {
   echo 'update done';
}
/* ... */
```

We finally implement this Web application using the **wamp** package installed on a Windows XP SP2 operating system. This package provides a web server: **apache** 2.2.6, a **PHP** 5.2.5 interpreter and the **MySQL** 5.0.45. This application then listens on the IPv4 address IP1 port 80. Meanwhile we launch an instance of Nepenthes on a GNU/Linux Debian 4.0r2 operating system on IP2. From another IP address, we send the following request on a TCP socket to IP2 at TCP port 80.

```
``GETwget IP1/sqlinjection/updatePassword.php?newPass=pouet'\
/**/where/**/username='admin'/*''
```

When processing the request the application writes

```
"update users set userpassword ='pouet'/**/where/**/username=\
'admin'/*' where username='' and userpassword=''"
```

in the "$request" variable. Once the preprocessor used by the MySQL interpreter inside the mysql_query function removes the commentaries marked with the "/*", "*/" strings, the variable contains:

```
"update users set userpassword ='pouet' where username='admin'"
```

. Finally the execution of this SQL query creates the situation described in Figure 3.1: the the password of the "admin" user has changed from "aj6che:$" to "pouet".

3.1.2 Making PHP.HoP Launch a Shellcode Injection Attack

In this section we show how we make PHP.HoP generate a shellcode injection attack. By this experiment we show that PHP.HoP can be used as an intrusion proxy to make the vulnerable application execute the code of our choice, like the download and execution of a malware. For visual reasons, we choose in this case, to make it execute the "calc.exe" program which creates a new window unlike malware which try to be more furtive.

The way download requests are computed by both honeypots implies to use *alphanumeric shellcodes* [213]. *Alphanumeric shellcodes* are polymorphic shellcodes that rely on instructions that can be expressed using alphanumeric bytes, i.e more or less all uppercase and lowercase letters, together with digits. Some available software [22] provides solutions to build such `alphanumeric shellcodes` when targeting applications using Windows SDK on `x86-32` von Neumann machines [15]. That is why we choose to develop our vulnerable application for this target. We therefore obtained an alphanumeric shellcode that executes the **calc.exe** process. However the problem of transmitting a valid alphanumeric address that references instructions resulting on the execution of our injected `shellcode` remains.

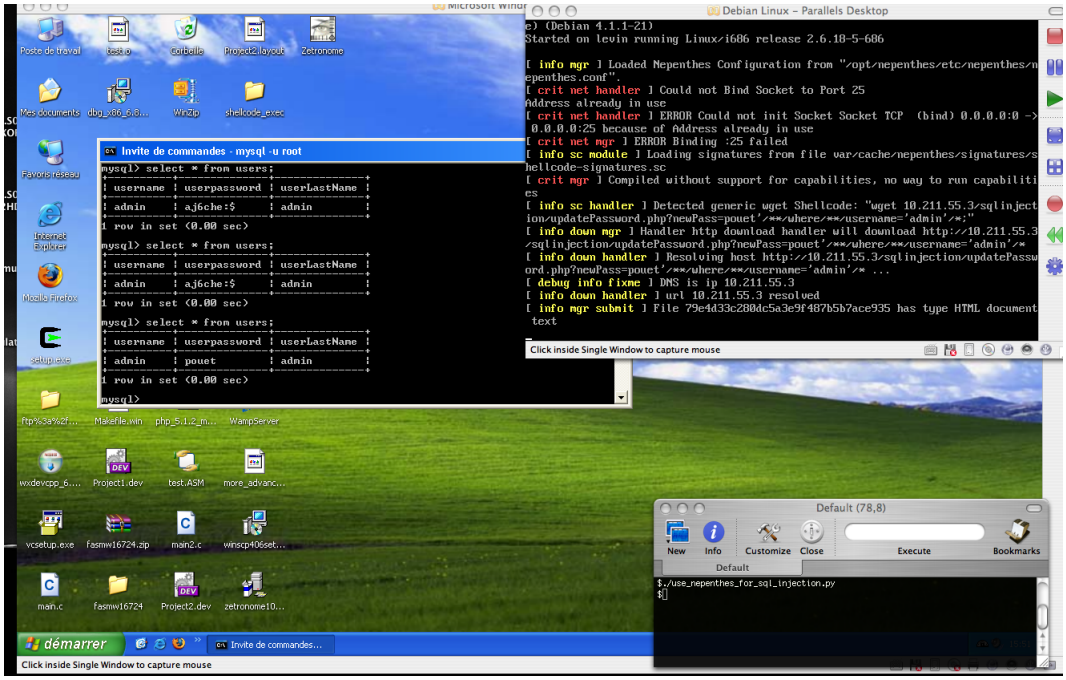


Figure 3.1: Making Nepenthes launch an SQL injection.

To overcome this issue we have purposely injected a **jmp %esp** instruction[2] at an alphanumerical address on an executable page of the vulnerable process address space as described by the following code:

```
/* ... */
unsigned char jmp_esp[2]={ 0xff, 0xe4 } ;
/* ... */

unsigned long int new_addr;
new_addr = (unsigned long int)VirtualAlloc(NULL,
                                           0x20204040,
                                           MEM_COMMIT,
                                           PAGE_EXECUTE_READWRITE);
addr = new_addr + 0x20203030;
availableAddrNotFound = !check_addr_available(addr);
memcpy((void*)addr,jmp_esp,2);
```

The `VirtualAlloc` function of the Windows SDK enables one process to obtain a new page of the size of our choice with the `READ,WRITE,EXECUTE` rights. The idea behind our heuristic is to allocate a sufficient big memory region, so that we can find one address that could be expressed in the range allowed by the alphanumeric filter [3].

[2] Which executes what is on top of the stack.
[3] Which is what check_addr_available checks.

This heuristic worked and gave us a memory region of value 0x30523030, which in little endian and ASCII, gives “00R0”. Now we know we have our `jmp_esp` in this address. Then we developed a buggy web server vulnerable to a stack smashing exploit as described in Listing 3.3.

Listing 3.3: Web server code

```
/* ... */

#define GET_PREAMBLE_SIZE 5
#define GET_REQUEST_SIZE 1
#define HTTP_REQUEST_MAX_SIZE 1000


int handleGetRequest(char* request) {
  char buff[GET_REQUEST_SIZE];
  strcpy(buff,request);
  return 1;
}

int handleNewClient(SOCKET* pClient) {
  char incomingBuff[HTTP_REQUEST_MAX_SIZE];
  int nberReadAll=0,nberRead=0;
  nberReadAll = recv(*pClient,incomingBuff,HTTP_REQUEST_MAX_SIZE,0);
  incomingBuff[nberReadAll]=0x00;//end of string
  /* ... */
  if (memcmp(incomingBuff,"GET /",GET_PREAMBLE_SIZE)==0) {
    printf("GET request");
    printf(NEW_LINE);
    return handleGetRequest(incomingBuff+GET_PREAMBLE_SIZE);
  }
  /* ... */
}



int _tmain(int argc, _TCHAR* argv[]){
  /* ... */
  while(1) {
    if ( (client = accept(sd, (sockaddr*)&client_addr,
                          &sizeofSockAddrIn)) == -1) {
      break;
    }
    handleNewClient(&client);
    closesocket(client);
  }
  /* ... */
}
```

There the `strcpy` function in `handleGetRequest` will obviously smash the `return address` if the request is too big. The details of a successful intrusion are illustrated in Figure 3.2.

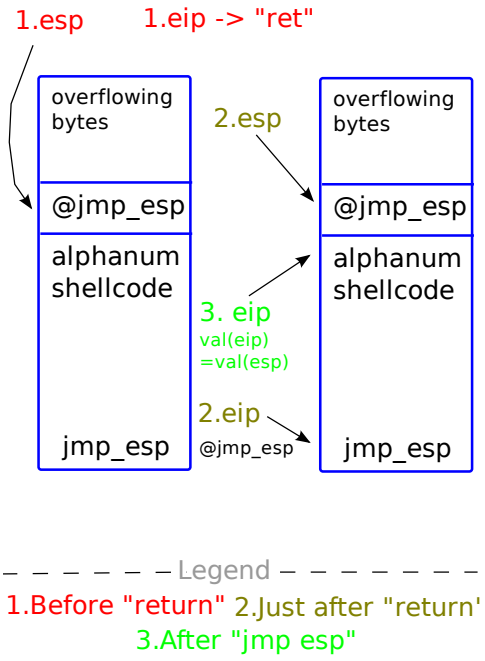


Figure 3.2: After overflowing the `return address` of the `handleGetRequest` function(1), when executing the corresponding **ret** instruction, our software actually executes the instructions located at the `jmp_esp` address (2), which will then execute our `alphanumeric shellcode` located on the top of the remaining stack (3). N.B: EIP: Extended Instruction Pointer, the PC we mentioned in Section 1.1.1.3 for this Von Neumann machine. ESP: Extended Stack Pointer [15].

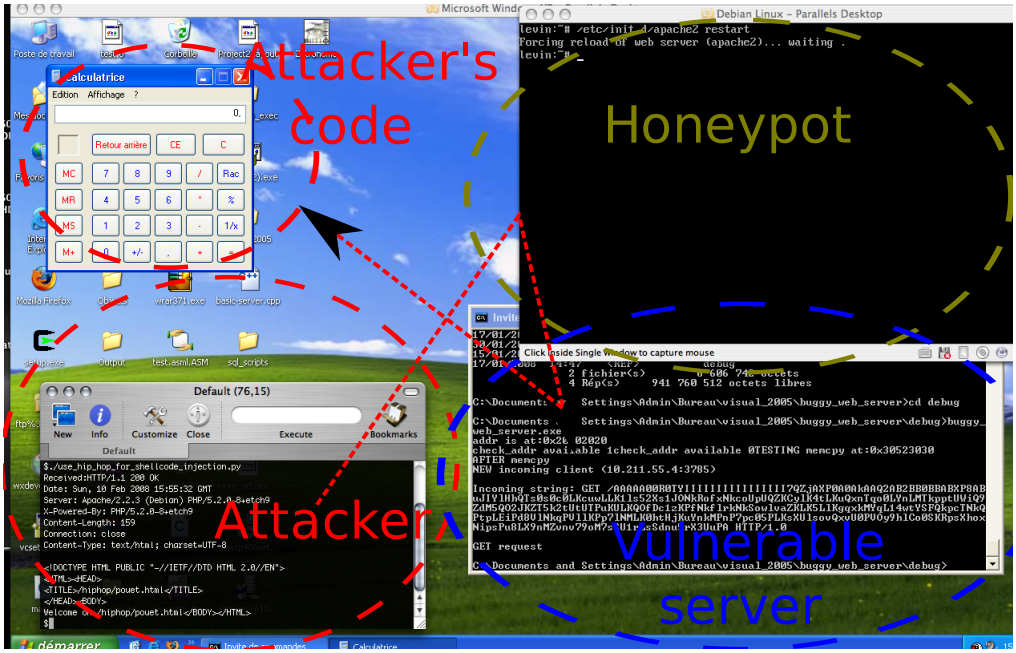


Figure 3.3: Making PHP.HoP launch a stack smashing exploit.

To implement this scenario, we rebuilt the same configuration than the past one, unless we changed Nepenthes with PHP.HoP and the **wamp** `http server` with our buggy server we compiled with visual studio 2005 [4]. Finally the result of the sending of the request:

```
shellcode="TYIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJI"+\
"YlHhQTs0s0c0LKcuwLLK1ls52Xs1JONkRofxNkcoUpUQZKCylK4tLKuQxnTqo0"+\
"LYnLMTkpptUWiQ9ZdM5QO2JKZT5k2tUtUTPuKULKQOfDc1zKPfNkflrkNkSowl"+\
"vaZKLK5LlKgqxkMYqL14wtYSFQkpcTNkQPtpLEiPd8VlNkqPVllKPp7lNMLK0h"+\
"tHjKuYnkMPnP7pc05PLKsXUlsovQxvU0PVOy9hlCo0SKRpsXhoxNipsPu8LX9n"+\
"MZvnv79oM7sSU1rLsSdnu5rX3UuPA"
jmp_esp="00R0"
request="GET /hiphop/pouet.html?wget%20http://10.211.55.3/"+"A"*5+\
  jmp_esp+shellcode+"; HTTP/1.0\r\n\r\n"
```

to the PHP.HoP instance can be seen in Figure 3.3: the `calc.exe` was successfully executed.

Even if such attacks, i.e. use a honeypot as an attack proxy-server, have not been seen in the Internet, we do think it can be worthy to make the exercise of monitoring such communications.

[4] Of course disabling security cookies.

3.2 Design of Necessary Components to Monitor Malware Downloading Honeypots

This section aims at showing how we use Luth to implement an acceptable compromise between dropped and accepted packets when monitoring the Nepenthes honeypot's network communications. When inspecting the involved packets, we differentiate two different situations:

- Sessions towards the servers implemented by the honeypot, like intrusion attempts.
- Sessions where the honeypot implements clients talking to third party servers, like some download requests.

In the first case, malicious client packets should be accepted to allow the intrusion emulation. In the second case, malicious client packets should be dropped, to prevent the attacking of third-party ADPS.

The first section illustrates how we implement this compromise when analyzing TCP session establishment segments. Indeed some of the intrusion attempts emulated by the honeypot contain malicious TCP segments that must be accepted. However, we should prevent the malicious TCP segments from download requests. Therefore a section deals with the two different TCP MI we developed: MTCP that allows some malicious segments, and TCP that implements a non malicious midpoint TCP stack.

The last section shows how we monitor the client download requests made by the honeypot by describing the Ftp, Http, Tftp and Link MI-s.

3.2.1 TCP and MTCP MI-s

Malware trying to attack these honeypots need first to detect the presence of a potentially vulnerable service. To do so, most of the time, they perform an exhaustive research on some part of the IP address space. This can be done for example, by starting with the first address, trying the second, etc. However, all the deduced IP addresses do not host a potentially vulnerable service. Therefore, before trying to launch an attack, it could be interesting to reduce the number of addresses to attack. This job is usually called *scanning*. The need for *scanning* can be justified by two arguments:

1. Deducing a host is potentially vulnerable is computed faster than performing the intrusion and checking it worked. Taking into account that the number of addresses to try is consequent, this strategy greatly reduces the time to perpetrate successful intrusions on all the vulnerable servers in the chosen IP address space.
2. Considering that some systems could detect the launching of such an attack, it is more furtive to reduce the number of launched intrusions.

The way endpoints protocols should or must behave in the Internet is specified by means of Request For Comments (RFC) documents. The way the TCP protocol should behave is therefore described in an RFC too [35]. Most of the time, for performance reasons, such scanning is performed by talking in non RFC compliant ways with the potential TCP server on the scanned IP address. This creates a problem for the design of TCP MI-s. Indeed, an RFC compliant TCP MI drops such non compliant messages and therefore, prevents such messages to reach the scanned TCP server. However, without scanning segments, the honeypot will not be selected in the addresses to send the intrusion. Therefore, the honeypot will not emulate the behaviour of a successfully attacked server, and will not download the related malware. However, a TCP MI that allows scanning patterns will make a non honeypot server

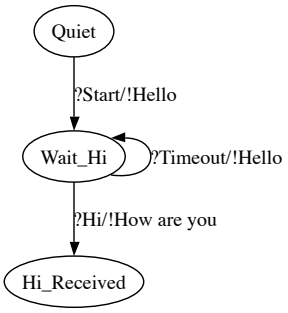


Figure 3.4: Hello Mealy Machine

visible to such malware. That's why we chose to design two different MI-s, the RFC compliant one TCP MI, and the one allowing scanning messages we call Malicious TCP MI (MTCP).

The difference in both MIs resides in the way they monitor TCP sessions establishments.

3.2.1.1 TCP MI Automata

Let us illustrate the way TCP establishes and closes its connections according to its standard mealy machine depicted in the RFC 793. A mealy machine is a six-tuple $(Q, \Sigma, \Gamma, \delta, \gamma, q_0)$, where:

- Q is a set representing the states of the machine,
- $q_0 \in Q$ a start state,
- Σ a set called input alphabet,
- Γ a set called output alphabet,
- δ is a transition function defined in $Q \times \Sigma \rightarrow Q$,
- γ is an output function defined in $Q \times \Sigma \rightarrow \Gamma$.

The machine $(\{Quiet, Wait\_Hi, Hi\_Received\}$,
$\{Start, Timeout, Hi\}$,{Hello,How are you},γ_s,δ_s,$Quiet)$ with

- $\gamma\_s(Quiet, Start) = Wait\_Hi$,
- $\delta\_s(Quiet, Start) =$ Hello,
- $\gamma\_s(Wait\_Hi, Hi) = Hi\_Received$,
- $\delta\_s(Wait\_Hi, Hi) =$ How are you,
- $\gamma\_s(Wait\_Hi, Timeout) = Wait\_Hi$,
- $\delta\_s(Wait\_Hi, Timeout) = Hello$,

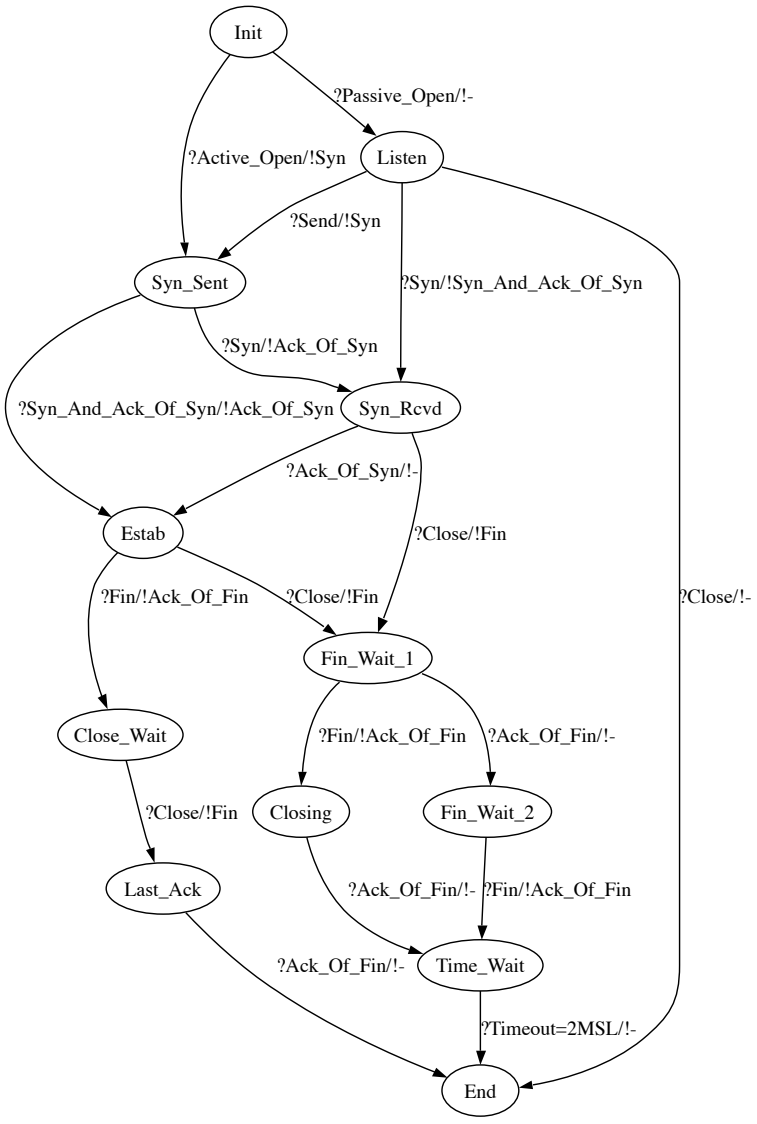


Figure 3.5: TCP standard

is depicted in Figure 3.4. The machine in Figure 3.5 is the one depicted in TCP's RFC[5]. The transitions on this machine are of two kinds:

- network messages: $\{Syn, Ack\_Of\_Syn, Syn\_And\_Ack\_Of\_Syn, Fin, Ack\_Of\_Fin\}$,
- non network messages, which are primitives used to give an upperlayer a way to interact the TCP stack: $\{Passive\_Open, Active\_Open, Send, Close\}$ together with a timeout event $\{Timeout = 2MSL\}$.

This machine, even if designed for a client/server model, models both clients and servers behaviours. Indeed, this can be seen as an early step towards nowadays popular peer2peer communication models, where no distinction is done among the entity sending the first message, the client, and the one waiting for messages, the server. To inspect TCP connections setups, the most reasonable solutions for the *TCP* MI is to use a mealy machine too. To make a distinction between client and server's messages, we prefix each messages sent by their senders name, i.e. instead of calling a *Syn* sent by a client, we call it *Client_Syn*. To have an understanding of the situations that can arise, let us take the example of a client trying to establish a connection to a server, which is asked to close before having reached the ESTABLISHED state as depicted in Figure 3.6. To do so, among others, we need to answer the following question: after having forwarded a *Client_Syn* should the midpoint drop a *Server_Fin* coming from the server ? To know what Netfilter, the framework implementing the firewall of the GNU/Linux operating system thinks about it, we use the Scapy packet forging tool [42] to create the client and server software depicted in Listings 3.5 and 3.4.

Listing 3.4: Scapy server

```
from scapy import *
def tcp_monitor_callback(pkt):
    tcp=pkt.getlayer('TCP')
    ip=pkt.getlayer('IP')
    if ip and tcp and (tcp.dport==2002):
        nseq=1
        nack=tcp.seq+1
        nsrc=ip.dst
        ndst=ip.src
        nsport=tcp.dport
        ndport=tcp.sport
        ip1=IP(src=nsrc,dst=ndst)
        tcp1=TCP(sport=nsport,dport=ndport,seq=nseq+1,ack=nack,flags='F')
        ip2=IP(src=nsrc,dst=ndst)
        tcp2=TCP(sport=nsport,dport=ndport,seq=nseq,ack=nack,flags='SA')
        send(ip1/tcp1)
        send(ip2/tcp2)

def run():
    sniff(iface="eth1",prn=tcp_monitor_callback,store=0,count=10)

run()
```

[5]To have a more easily understandable view of the machine, we have split the CLOSED state into Init and End state

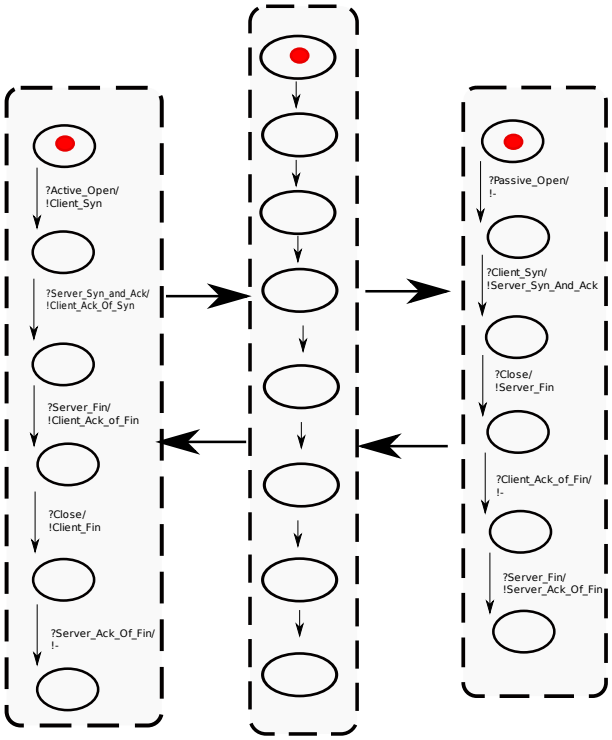


Figure 3.6: TCP midpoint automata.

Listing 3.5: Scapy client

```
#! /usr/bin/env python
import sys
from scapy import *

def tcp_monitor_callback(pkt):
    tcp=pkt.getlayer('TCP')
    if tcp:
        print tcp.summary()

def run():
    send(IP(dst="192.168.10.16")/TCP(dport=2002,flags='S'))
    sniff(iface="eth3",prn=tcp_monitor_callback,store=0,count=2)

run()
```

The client written in Scapy sends a *Client_Syn* to the server in 192.168.10.16 on port 2002 and waits for incoming segments. The server, after having received the *Client_Syn*, sends a *Server_Fin*, and then a *Server_Syn_And_Ack_Of_Syn*. The two endpoints are interconnected by a router[6], which first forwards all the packets. The observation of the packets received by the client using the tcpdump traffic dumper gives the following result:

```
11:26:27.595902 IP 192.168.2.9.ftp-data > 192.168.10.16.2002: S 0:0(0)
11:26:27.625104 IP 192.168.10.16.2002 > 192.168.2.9.ftp-data: F 2:2(0)
11:26:27.661081 IP 192.168.10.16.2002 > 192.168.2.9.ftp-data: S 1:1(0) ack 1
```

The other experiment is conducted by asking the Netfilter firewall to inspect, among others, the TCP connection establishments. This firewall has chosen to encode all the different protocol statefullness into a single set of tokens: *ESTABLISHED*, *NEW*, *RELATED* and *INVALID*. The *INVALID* token means that a packet does not match a normal behaviour of the protocol stack. The first rule we implement is the following:

```
*filter
:INPUT ACCEPT [1188072:148255333]
:FORWARD DROP [1:40]
:OUTPUT ACCEPT [1261742:197788100]
-A FORWARD -i eth1 -o eth0 -p tcp -m tcp --tcp-flags SYN SYN -j ACCEPT
-A FORWARD -i eth0 -o eth1 -p tcp -m state
   --state INVALID,NEW,RELATED,ESTABLISHED -j ACCEPT
```

We launch the client and servers again and observe that the same packets are received by the client:

```
11:26:27.595902 IP 192.168.2.9.ftp-data > 192.168.10.16.2002: S 0:0(0)
11:26:27.625104 IP 192.168.10.16.2002 > 192.168.2.9.ftp-data: F 2:2(0)
11:26:27.661081 IP 192.168.10.16.2002 > 192.168.2.9.ftp-data: S 1:1(0) ack 1
```

[6]There is no system-call available the GNU/Linux operating system to shutdown the TCP stack. Therefore, even if the segments sent with Scapy are grabbed by the "sniff" function, the TCP stack will receive them too. Moreover, because no application has been registered to handle the messages related to the TCP server port 2002 by means of "bind" and "listen" system-calls, the TCP stack will generate an RST segment after receiving all the messages sent by the other host with Scapy. Therefore, so that the TCP stack of both client and server does not send these RST segments, we add iptables rules in both endpoints to DROP them. For example in the server: iptables -A OUTPUT -o eth1 -p tcp -m tcp –sport 2002 –tcp-flags RST RST -j DROP.

We configure then the following second Netfilter rule that drops the previously accepted packets considered as INVALID (by suppressing the INVALID state in the state option of the rule).

```
*filter
:INPUT ACCEPT [999917:124200639]
:FORWARD DROP [1:40]
:OUTPUT ACCEPT [1071348:170388728]
-A FORWARD -i eth1 -o eth0 -p tcp -m tcp --tcp-flags SYN SYN -j ACCEPT
-A FORWARD -i eth0 -o eth1 -p tcp -m state
   --state NEW,RELATED,ESTABLISHED -j ACCEPT
```

We launch for a last time, the client and servers again and we observe the packets received by the client:

```
11:29:30.145213 IP 192.168.2.9.ftp-data > 192.168.10.16.2002: S 0:0(0)
11:29:30.241349 IP 192.168.10.16.2002 > 192.168.2.9.ftp-data: S 1:1(0) ack 1
```

We can see that the *Server_Fin* segment has been dropped by Netfilter, meaning that it considers this packet as *INVALID*. However, as depicted in [78], this arbitrary choice can be discussed. We should remind that as described in Figure 2.4, due to possible packet loss and reorderings, the sequence of packets seen by midpoints are different from the one received by the endoint. Indeed, by quoting the Figure 3.6, after receiving the *Client_Syn*, the server sends a *Server_Syn_And_Ack*. Let us say the server receives a *Close* event and therefore sends a *Server_Fin*. *Server_Fin* and *Server_Syn_And_Ack* are now in the network. Let us imagine Internet reorders both messages before reaching the midpoint. That way midpoint sees *Server_Fin* before *Server_Syn_And_Ack_Of_Syn*, the same situations we presented using the client and server written in Scapy. Let us say that unlike the situation arose with the second Netfilter configuration rule, the midpoint forwards them. *Server_Fin* and *Server_Syn_And_Ack_Of_Syn* are in the way to the client. Let us say the network reorders them once again, and that the latter receives *Server_Syn_And_Ack_Of_Syn* and *Server_Fin*. In this situation, the client continues its standard behaviour, which argues for a positive answer to the first question: for interoperability reasons, the midpoint should forward a *Server_Fin* after having forwarded a *Client_Fin*.

The skeptical reader could think this double reordering is a very peculiar case which will probably never happen in the Internet. For these reasons, one could say that it could reasonably be neglected. We'd like here again to make an analogy with similar reasonings that conducted to the developing of software bugs. Let us say a computer scientist writing the lines of a program that reads a username from an input accessible by any one, starts thinking: "Do I have to check the boundaries of the buffer when writing the input username on it ? What kind of person will enter a name with more than 1024 letters ? Well probably no one, so I'd better not make the complicated verifications related to the boundaries of my buffer and focus my work on other "real" problems ." Indeed, the great majority of developers (me included), before understanding the functioning of "buffer overflow" code injection attacks, make similar reasonnings. But not taking these a priori improbable cases into account, makes some software vulnerable to intrusion attacks, which makes us strongly question the validity of such argumentation.

Moreover, the reader should notice that this peculiar case has been found by manually exploring one of the cases computed by the solution we present later. Perhaps, the a-priori scarcity of such situations will be questioned too after counting the number of such unlikely situations.

To sum up, the problems illustrated by the inspection of TCP connection establishment is just a particular case of the side effects an arbitrarily computed midpoint inspection algorithm can create. For example in [214], Van Rooij showed that neglecting these problems when inspecting TCP's control flow algorithm from a midpoint, conducts to deadlock-ed TCP sessions. That's why instead of computing

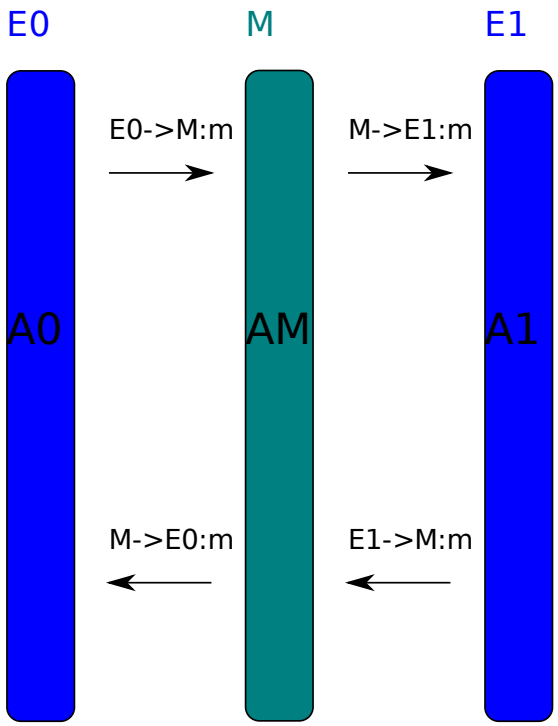


Figure 3.7: Model of inspection actors.

arbitrary TCP mealy machines, we have used the generic solution proposed by Bidder-Senn et. al. [78]. The latter study proposes to synthesize the midpoint mealy machine, by analyzing two Mealy machines, which model endpoints behaviour.

As depicted in Figure 3.7, in this model the end hosts E_0, E_1 are interconnected to the midpoint M by four different pipes. The behaviour of both endpoints is driven by two Mealy machines, one for E_0, called $A_0 = (Q_0, \Sigma_0, \Gamma_0, \delta_0, \gamma_0, q_{0,1})$ and another for E_1, called $A_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, \gamma_1, q_{1,1})$ with $\Sigma_0 = \Gamma_1$, $\Sigma_1 = \Gamma_0$. The alphabet of both input and output messages are the network messages they send and receive, i.e. *Client_Syn*, *Server_Fin*, *Server_Syn_And_Ack_Of_Syn* ..., together with the empty element '$-$'.

Definition 8 *The empty element '$-$' models:*

- *a transition based on non network events, like timeouts, software calls,...*
- *a transition that does not output a network message.*

The endpoint tracks the messages in the network with a multiset $net \subseteq \wp(S)$ where $S = \{X \rightarrow Y : m | X, Y \in \{E_0, E_1, M\}, X \neq Y, m \in (\Sigma_0 \cup \Gamma_0)\}$ represents the messages that transit in the four pipes. The automata of the midpoint, which decides whether to forward or drop a packet, is

synthesized first by guessing in what states both the endpoint and the network are. Then, it chooses to forward only the messages that were predicted. Going back to mathematics, the midpoint mealy machine is constructed like this:
$A_M = (Q_M, \Sigma_M, \Gamma_M, \lambda_M, \delta_M, q_{0,M})$
$Q_M = P(Q_0 \times Q_1 \times net)$
$\Sigma_M = \{E_0 \to M : a | a \in (\Gamma_0 \backslash \{-\})\} \cup \{E_1 \to M : a | a \in (\Gamma_1 \backslash \{-\}\}$
$\Gamma_M = \{M \to E_0 : a | a \in (\Sigma_0 \backslash \{-\})\} \cup \{M \to E_1 : a | a \in (\Sigma_1 \backslash \{-\}\} \cup \{-\}$
$q_{M,1} = \{(q_{0,1}, q_{1,1}, \{\})\}$

The δ_M function, depends of the following notion

$$succ(q_m) = \bigcup_{q \in q_M} \bigcup_{(M \to E_0 : m_1) \in net_M} \bigcup_{(M \to E_1 : m_4) \in net_M} \bigcup_{msg \in net_M}$$

$$\{(q_0, q_1, net_M), \quad \text{Nothing happened yet} \quad (3.1)$$
$$(q_0, q_1, net_M \backslash \{msg\}), \quad \text{Message lost} \quad (3.2)$$
$$(\delta_0(q_0, m_1), q_1, (net_M \backslash \{M \to E_0 : m_1\}) \cup m_2), \quad E_0 \text{ takes one message} \quad (3.3)$$
$$(\delta_0(q_0, -), q1, net_M \cup m_3), \quad \text{Empty transition of } E_0 \quad (3.4)$$
$$(q_0, \delta_1(q_1, m_4), (net_M \backslash \{M \to E_1 : m_4\} \cup m_5), \quad \text{Same as for } E_0 \quad (3.5)$$
$$(q_0, \delta_1(q_1, -), net_M \cup m_6\} \quad \text{Idem that for } E_0 \quad (3.6)$$

Where

$$m_2 = \begin{cases} \{E_0 \to M : \lambda_0(q_0, m_1)\} & \lambda_0(q_0, m_1) \neq -, \\ \emptyset & \text{otherwise} \end{cases}$$

$$m_3 = \begin{cases} \{E_0 \to M : \lambda_0(q_0, -)\} & \lambda_0(q_0, -) \neq -, \\ \emptyset & \text{otherwise} \end{cases}$$

$$m_5 = \begin{cases} \{E_1 \to M : \lambda_1(q_1, m_4)\} & \lambda_1(q_1, m_4) \neq -, \\ \emptyset & \text{otherwise} \end{cases}$$

$$m_6 = \begin{cases} \{E_1 \to M : \lambda_1(q_1, -)\} & \lambda_1(q_1, -) \neq -, \\ \emptyset & \text{otherwise} \end{cases}$$

This succ function computes all the events (packet losses, one packet taken as input, empty transition) that could have occurred during one endpoint transition. Then as multiple messages could be in the network, multiple transitions could occur producing different network and endpoint states. That's why instead of $succ(Q_M)$,

$$cl(succ(q_M)) = \bigcup_{i=0}^{i=\infty} succ^i(q_M)$$

is computed which handles packet reordering.

Then, the rest of the work consists in forwarding a message only if its apparition has been predicted by one of the states in $cl(succ(q_M)$. The successor state of the midpoint transition upon a message arrival is the union of all states that predicted the apparition of this message, with their net appended with the latter going to the other direction. In other words:

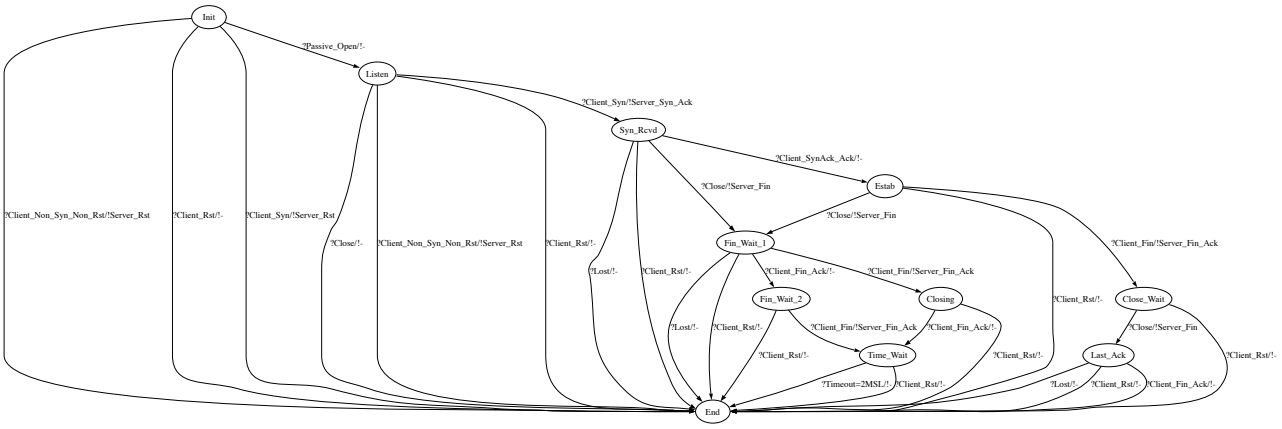


Figure 3.8: Model of the TCP server.

$$out((q_0,q_1,net_M),E_i \to E_M : y) = \begin{cases} (M \to E_{1-i} : y) & \text{if} \{E_i \to M : y\} \in net_M, \\ - & \text{otherwise} \end{cases}$$

$$\lambda_M(q_M,m) = \begin{cases} out(q,m) & \text{if } \exists \text{ q } \in cl(succ(q_M)) \mid out(q,m) \neq -, \\ - & \text{otherwise} \end{cases}$$

and

$$\delta_M(q_M,m) = \bigcup_{(q_0,q_1,net_M) \in cl(succ(q_M))} \{(q_0,q_1,(net_M \backslash \{m\}) \cup \lambda_M(q_M,m))\} \mid m \in net_M\}$$

Using this model, computing RFC compliant midpoint TCP-automata is straightforward. We have chosen not to take the exact Mealy-machine presented in the RFC of TCP (see Figure 3.5) for two reasons. First this machine does not differentiate client and servers as it models both behaviours. If we put this machine in input of the midpoint automata computing algorithm, two TCP clients establishing a session and communicating will be allowed. However, our case study only deals with client and servers, we have therefore chosen to split the generic automata into two automatas, one representing the client and the other representing the server. Second, the RST messages, whose use is described by a list of recommendations, are not modeled by the standard mealy machine (see Figure 3.5) . However, such messages play a central role in real TCP sessions.

The client and server mealy machines we use are depicted in Figure 3.8 and 3.9. To model the fact that a closed TCP server sends an RST segment to any non RST or SYN messages, we add the *Client_Non_Syn_Non_Rst* segment to factorize all such messages in one. We incorporate into the *TCP* MI the resulting midpoint automata with the automatas of these TCP client and servers, but that sends one retransmission. We only printed the version of automatas without retransmission because the other figures do not fit in a single page. The algorithm to compute the midpoint automata from the client and servers with one retransmission needs more than 2 GB of memory. Therefore, we use a Sun X4600 M2[7] server present in our laboratory, with 208 GB of RAM shared among the eight quad-core processors to compute the automatas. We then export the computed results to other computers using OCaml's Marshal facility provided by the standard library. We stress this automata by inspecting the sessions

[7] http://www.sun.com/servers/x64/x4600/

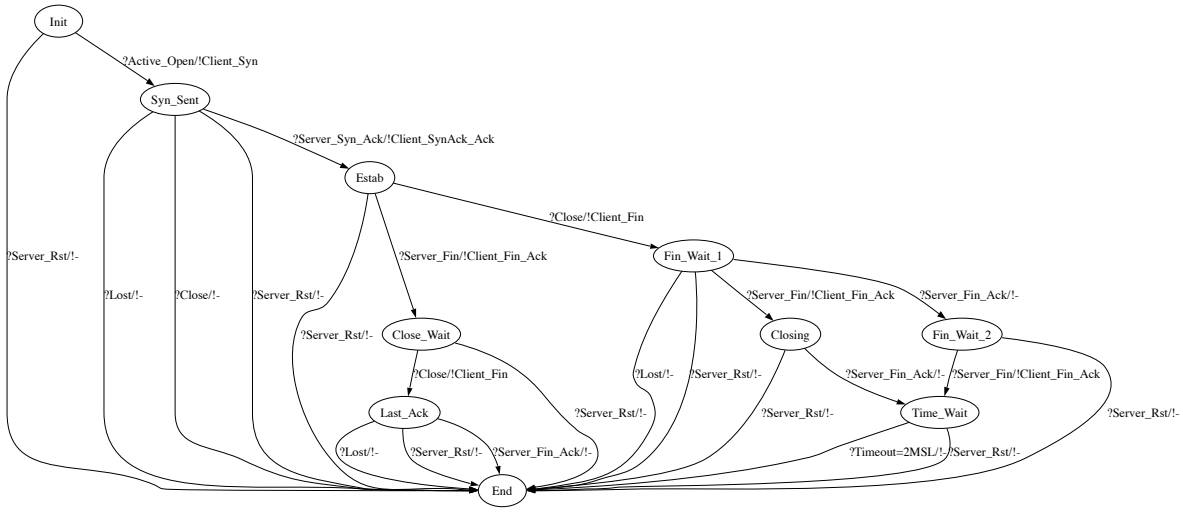


Figure 3.9: Model of the TCP client.

the Nepenthes honeypot establishes. As expected, we observe that quite a lot of TCP sessions towards honeypot servers, most due to scanning patterns, are dropped by this automata. This motivates the need to develop the *MTCP* MI, which has a different automata.

3.2.1.2 MTCP MI Automata

Among the previously dropped scanning segments, we identify five different patterns:

1. a client that sends an ACK, waits for a RST[8] and performs a three-way handshake then:

```
13:09:41.233609 IP cl_ip.2248 > h_ip.445: . ack ACKN
13:09:41.233658 IP h_ip.445 > cl_ip.2248: R ACKN:ACKN(0)
13:09:44.260410 IP cl_ip.2248 > h_ip.445: S SCLN:SCLN(0)
13:09:44.260429 IP h_ip.445 > cl_ip.2248: S SSRN:SSRN(0) ack SCLN+1
13:09:44.371707 IP cl_ip.2248 > h_ip.445: . ack SSRN+1
13:09:44.483669 IP cl_ip.2248 > h_ip.445: P SCLN+1:SCLN+52(51) ack SSRN+1
13:09:44.483682 IP h_ip.445 > cl_ip.2248: . ack SCLN+52
13:09:48.270413 IP cl_ip.2248 > h_ip.445: F SCLN+52:SCLN+52(0) ack SSRN+1
13:09:48.271015 IP h_ip.445 > cl_ip.2248: F SSRN+1:SSRN+1(0) ack SCLN+53
13:09:48.492884 IP cl_ip.2248 > h_ip.445: . ack SSRN+2
```

,

2. client that repetedly sends fin messages:

```
05:16:49.344739 IP cl_ip.33643 > h_ip.445: F FCLN:FCLN(0) ack FACLN
05:16:49.344753 IP h_ip.445 > cl_ip.33643: R FACLN:FACLN(0)
05:16:52.003775 IP cl_ip.33643 > h_ip.445: F FCLN:FCLN(0) ack FACLN
05:16:52.003783 IP h_ip.445 > cl_ip.33643: R FACLN:FACLN(0)
```

[8]Indeed, as a valid session does not start with an ACK, a legitimate server will answer a RST.

Pattern number	Path in the automata
1,2,3	*Init_0* → *Scan_non_syn_sent_0* → *End_0*
4	*Init_0* → *Scan_syn_sent_0* → *End_0*
5	*Init_0* → *Syn_Sent_0* → *Estab_0* → *End_0*

Figure 3.10: Mapping between scanning pattern numbers and part of the malicious TCP automata.

```
05:16:57.889275 IP cl_ip.33643 > h_ip.445: F FCLN:FCLN(0) ack FACLN
05:16:57.889284 IP h_ip.445 > cl_ip.33643: R FACLN:FACLN(0)
```

3. the same with SYN ACK instead of FINs,

```
09:03:37.115778 IP cl_ip.80 > sr_ip.22446: S SCLN:SCLN(0) ack SACLN
09:03:37.115802 IP sr_ip.22446 > cl_ip.80: R SACLN:SACLN(0)
09:03:40.823581 IP cl_ip.80 > sr_ip.22446: S SCLN:SCLN(0) ack SACLN
09:03:40.823589 IP sr_ip.22446 > cl_ip.80: R SACLN:SACLN(0)
```

4. a client that performs a two-way handshake, resets the session and performs a three-way handshake then:

```
15:31:18.675517 IP cl_ip.3376 > h_ip.445: S SCLN:SCLN(0)
15:31:18.675570 IP h_ip.445 > cl_ip.3376: S SSRNPRIME:SSRNPRIME(0) ack SCLN+1
15:31:18.721587 IP cl_ip.3376 > h_ip.445: R SCLN+1:SCLN+1(0)
15:31:21.605267 IP cl_ip.3376 > h_ip.445: S SCLN:SCLN(0)
15:31:21.605282 IP h_ip.445 > cl_ip.3376: S SSRN:SSRN(0) ack SCLN+1
15:31:21.714431 IP cl_ip.3376 > h_ip.445: . ack SSRN+1
15:31:21.715176 IP cl_ip.3376 > h_ip.445: P SCLN+1:SCLN+52(51) ack SSRN+1
15:31:21.715189 IP h_ip.445 > cl_ip.3376: . ack SCLN+52
15:31:28.698784 IP cl_ip.3376 > h_ip.445: F SCLN+52:SCLN+52(0) ack SSRN+1
15:31:28.699377 IP h_ip.445 > cl_ip.3376: F SSRN+1:SSRN+1(0) ack SCLN+53
15:31:28.808467 IP cl_ip.3376 > h_ip.445: . ack SSRN+2
```

.

5. Finally, to speed up session ending, some clients reset ESTABLISHED sessions sending an RST message, instead of using the standard process involving FIN messages like that:

```
06:47:59.962948 IP cl_ip.1044 > sr_ip.139: S SCLN:SCLN(0)
06:47:59.962959 IP sr_ip.139 > cl_ip.1044: S SSRN:SSRN(0) ack SCLN+1
06:48:00.047759 IP cl_ip.1044 > sr_ip.139: . ack SSRN+1
06:48:00.054255 IP cl_ip.1044 > sr_ip.139: P SCLN+1:SCLN+77(76) ack SSRN+1
06:48:00.054264 IP sr_ip.139 > cl_ip.1044: . ack SCLN+77
06:48:00.054372 IP sr_ip.139 > cl_ip.1044: P SSRN+1:SSRN+65(64) ack SCLN+77
06:48:00.120591 IP cl_ip.1044 > sr_ip.139: R SCLN+77:SCLN+77(0)
```

To allow these vulnerability scanning patterns, we model the non standard malicious client as depicted in 3.11. The patterns are represented by the part of the automatas described in the Figure 3.10. This automata is here again, the version without retransmissions.

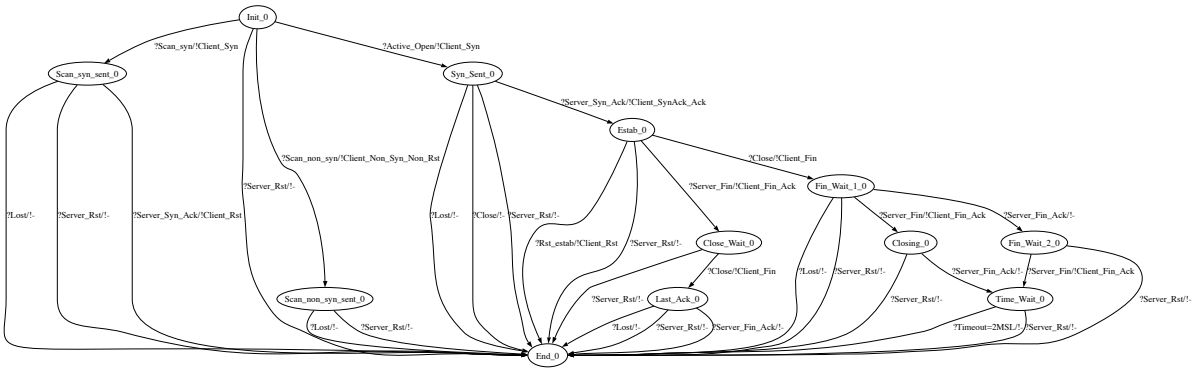


Figure 3.11: Model of the malicious TCP client.

The two transitions in Figure 3.11 going from *Init*_0 use three non network messages: ?*Scan_syn*, ?*Scan_non_syn* and ?*Active_Open*. However the model of the automatas used by Bidder-Senn et. al. can use a single non network transition per state: the '$-$' transition (Definition 8). For this reason the algorithm we use to compute the midpoint automata of a malicious TCP client and a standard TCP server contains the following modifications:

$$\Sigma_{network\ i} = \Gamma_{network\ 1-i}, i \in \{0,1\}$$
$$A_i = (Q_i, \Sigma_{non\_network\ i} \cup \Sigma_{network\ i}, \Gamma_i, \lambda_i, \delta_i, q_{0,i}), i \in \{0,1\}$$
$$\Gamma_M = \{M \to E_0 : a | a \in \Sigma_{network\ 0}\} \cup \{M \to E_1 : a | a \in \Sigma_{network\ 1}\} \cup \{-\}$$

$$succ(q_m) = \bigcup_{q \in q_M} \bigcup_{(M \to E_0 : m_1) \in net_M} \bigcup_{(M \to E_1 : m_4) \in net_M} \bigcup_{e_0 \in \Sigma_{non\_network0}} \bigcup_{e_1 \in \Sigma_{non\_network1}} \bigcup_{msg \in net_M}$$

$$\{(q_0, q_1, net_M), \quad \text{Nothing happened yet} \quad (3.7)$$
$$(q_0, q_1, net_M \backslash \{msg\}), \quad \text{Message lost} \quad (3.8)$$
$$(\delta_0(q_0, m_1), q_1, (net_M \backslash \{M \to E_0 : m_1\}) \cup m_2), \quad E_0 \text{ takes one message} \quad (3.9)$$
$$(\delta_0(q_0, e_0), q1, net_M \cup m_3), \quad \text{Non network trans of } E_0 \quad (3.10)$$
$$(q_0, \delta_1(q_1, m_4), (net_M \backslash \{M \to E_1 : m_4\} \cup m_5), \quad \text{Idem that for } E_0 \quad (3.11)$$
$$(q_0, \delta_1(q_1, e_1), net_M \cup m_6\} \quad \text{Idem that for } E_0 \quad (3.12)$$

Where

$$m_2 = \begin{cases} \{E_0 \to M : \lambda_0(q_0, m_1)\} & \lambda_0(q_0, m_1) \neq -, \\ \emptyset & \text{otherwise} \end{cases}$$

$$m_3 = \begin{cases} \{E_0 \to M : \lambda_0(q_0, e_0)\} & \lambda_0(q_0, e_0) \neq -, \\ \emptyset & \text{otherwise} \end{cases}$$

$$m_5 = \begin{cases} \{E_1 \to M : \lambda_1(q_1, m_4)\} & \lambda_1(q_1, m_4) \neq -, \\ \emptyset & \text{otherwise} \end{cases}$$

$$m_6 = \begin{cases} \{E_1 \to M : \lambda_1(q_1, e_1)\} & \lambda_1(q_1, e_1) \neq -, \\ \emptyset & \text{otherwise} \end{cases}$$

The automata used by the *MTCP* MI is the result of the combination of a malicious TCP client with one retransmission and the TCP server using one retransmission computed in the same Sun X4600 M2 server.

We have in this section depicted how we implement the compromise depicted in the beginning of Section 3.2 at the TCP level. However, we still need to implement the compromise in the protocols over transport protocols. We chose to inspect intrusion emulation by means of the *Void* MI that accepts them all[9]. If we use the *Void* MI to inspect the downloads requests, Luth will let the honeypot perpetrate the intrusion presented in Section 3.1. Therefore, the following section describes the MI used to drop the download requests that can contain code injections attacks.

3.2.2 Monitoring Download Requests Using Ftp, Http, Tftp and Link MI-s

Nepenthes downloads malware by using communication protocols where the honeypot is considered as:

- Server, using only TCP transport protocol.
- Client, using UDP for TFTP, and TCP for FTP, CRECEIVE, HTTP and LINK protocols. LINK is a non standard protocol used by some malware which consists in connecting to a server, sending four bytes, and then receiving the binary sent by the server. CRECEIVE is a protocol where the client receives the messages sent from the server without sending anything back. In other words, such a client downloads applicative data without uploading anything.

We monitor the protocols involving Nepenthes as a server by taking the same risks as when monitoring the vulnerabilities i.e. by the means of the *Void* MI. To monitor downloads where the honeypot acts as a client, we develop the following MI-s.

The CReceive MI drops all client messages. The Link MI is used to verify that the client only sends four bytes to the server. Indeed, in these four bytes, the intruders encodes a token used to authenticate the malware downloading request. As we do not see a way to perpetrate some kinds of attacks sending four bytes in a TCP session, we perform little filtering there.

The Tftp MI checks the performing of a $Tftp$ download[10], without upload. Here again, only the RFC compliance of the download is checked. The constraints written on this RFC are restricted enough so that we don't see how an attack could occur against the $Tftp$ server during the download of a file.

The Http(check_host=nepenthes) MI, checks that the HTTP header contains the options sent by Nepenthes. These informations are static and can therefore be easily verified. However, the content inside the GET request is dependent of the information sent by the intruder. To prevent the presence of SQL and shellcode injection attacks, the filter we use only accepts 30 bytes between two slashes $'/'$. Then among these 30 bytes, the only bytes accepted are in the character set

$$a\text{–}zA\text{–}Z0-9\backslash.\sim\_-$$

, i.e. alphanumeric bytes, plus the ., , _ and $-$ characters. This filter is just the result of a subjective point of view that states: a working remote code injection attacks would be difficult to develop in such conditions.

The MI $Ftp(ftpt = active)$ checks for the RFC compliance of the FTP session done by the Nepenthes client. Indeed, the honeypot makes an active Ftp. The only liberty given by the Nepenthes download parameters is the url of the file to download, the one used in the RETR command. We perform in this url the same filtering as in the strings contained between two '/' in the HTTP request.

[9] The risks related to this decision are depicted in Section 3.3.

[10] Only RRQ requests are allowed.

3.3 Experimentation

To evaluate the side effects of LUTH, we can perform either an online or an offline analysis (Section 2.3.2).

The online analysis consists in comparing the results of a non monitored and monitored honeypot. However to make a pertinent comparison, we need to assume some kind of periodicity on malware behaviour. Indeed, the results of both experiments depend of the side effects created by LUTH's inspection **and** of the behaviour of malware clients. These latter are however, up to our knowledge, unpredictable, and therefore, difficult to isolate from the results of such a comparison.

The offline analysis consists in analyzing a traffic dump representing the network activities of an unmonitored honeypot as depicted in Figure 3.12. Extrapolating the results obtained by an offline analysis can be criticized too. Indeed, the dropping of some packets, like for example a scanning pattern, can prevent the apparition of an intrusion attempt. During an online analysis, dropping scanning packet makes the attacker think the targeted honeypot does not exist. Therefore, instead of launching the intrusion, the latter will scan the next host. In this case, Luth will make the honeypot miss an opportunity to obtain a malware. During an offline analysis, Luth says whether it would have accepted or dropped packets present in the traffic dump. If we apply the previous situation to the offline analysis, Luth can report it dropped the scanning pattern and accepted all packets related to the intrusion and malware download that are in this case present in the traffic dump. However, considering that in this situation Luth only dropped the scanning pattern is false. Indeed, as shown by the online example, the packets related to the intrusion and download are virtually dropped when dropping the scanning pattern. Therefore, when performing an offline analysis, the side effects of Luth's dropping decisions are hard to extrapolate. The offline analysis permits however to analyze more efficiently LUTH's configuration and inspections, as the experiment can be performed as many times as wanted. That's why we finally chose the offline analysis paradigm.

This offline analysis aims at measuring the number of dropped packets and the related performance of the inspection performed by the rule depicted in Listing 3.6.

Listing 3.6: Monitoring Nepenthes

```
INSP([Udp(tm=30s,check_checksum=no)/[
        UPort(cl_addr=honeypot_address,sr_port=53)/Dns | (*L1*)
        UPort(cl_addr=honeypot_address,sr_port=69)/Tftp] (*L2*)
    |MTcp(tm=2min,check_checksum=no)/
        TPort(cl_addr=not honeypot_address,sr_addr=honeypot_address)/
        Void (*L3*)
    |Tcp(tm=2min,check_checksum=no)/
        TPort(cl_addr=honeypot_address,sr_addr= not honeypot_address)/[
        CReceive|Link|Http|Ftp] (*L4*)
  ]);;
```

3.3.1 Number and Justification of Dropped Packets

The first analyzed traffic dump represents a single week activity of a Nepenthes instance executed in one of the servers in our laboratory depicted in Figure 3.12. We have used this dump to make a detailed analysis of the causes of LUTH's drop decisions.

We aim with the rule depicted in Listing 3.6, to authorize as much packets while dropping the unwanted behaviours we identified. The latter rule is quite simple, it lets all the clients talk to the honeypot server on all possible TCP ports (L3). Then it monitors the TCP client requests made by the honeypot

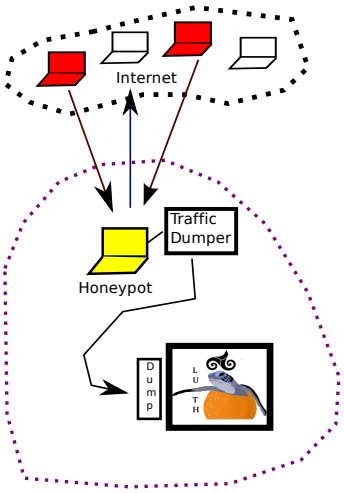


Figure 3.12: Offline analysis of honeypot network communications.

using the authorized malware downloading protocols (L4,L2). Finally it inspects the DNS requests made by the honeypot (L1).

The offline analysis reports that among the 88116 packets containing 13299789 bytes analyzed, 1913 are dropped (2.17 %). If we put a TCP MI put instead of the MTCP one in L3, 7453 packets are dropped (8.46 %). We investigate the reasons behind the dropping of the packets belonging to the 2.17 %, by analyzing first dropped UDP datagrams, and then dropped TCP segments.

3.3.1.1 Dropped UDP datagrams

The dropped UDP sessions are due to intrusion attempts targeting services over UDP emulated by the honeypot. To demonstrate the following fact, we can see that two types of UDP sessions are dropped by the rule depicted in Listing 3.6:

- Non DNS, TFTP sessions where the honeypot is behaving as a UDP client (UDP1),
- all the sessions where the honeypot behaves as a UDP server (UDP2).

We add the following rule, that accepts all the sessions of type UDP2, under L4:

```
| Udp(tm=2min)/
  UPort(cl_addr=not honeypot\_address,sr_addr=honeypot\_address)/
  Void (*L5*)
```

.

With this new rule Luth does not drop UDP packets anymore. Therefore, the dropped UDP sessions are only part of the UDP2 group. Moreover, this new rule reduces the number of drop decisions to 1620 (1.83 %). However, we prefer not to use this rule for the following reason. The UDP protocol does not have a session in it and, among others, permits one endhost to impersonate or spoof other IP addresses. Let us consider the scenario where one malicious endhost perpetuates a successful real (i.e. non emulated) intrusion on the vulnerable honeypot software. Let us imagine the latter wants then to

launch another intrusion from this hijacked honeypot against a new target, *nt* using the UDP protocol. Of course, Luth must drop this intrusion attempt. With L5 under L4, LUTH only accepts the following UDP sessions:

1. from the honeypot to a DNS server, where a quite aggressive DNS inspection is performed, (L1)
2. from the honeypot to eventual TFTP servers, where a quite aggressive TFTP inspection is performed, (L2)
3. from another endhost to the honeypot. (L5)

From now on, we use the notation $cl \rightarrow_{UDP} sr$ to illustrate a UDP datagram sent from cl to the server sr.

By using a partner in the Internet, this attacker could spoof the address of the new target *nt*, to send a UDP datagram towards the corrupted honeypot, *hp*, i.e $nt \rightarrow_{UDP} hp$. LUTH's inspection would accept it and create a pseudo-state saying there is a communication between the client with the address of the spoofed target *nt*, and the hijacked honeypot server, with the address *hp*. The intruder on the honeypot could then launch the wanted attack on the targeted server, $hp \rightarrow_{UDP} nt$, without being dropped by LUTH's inspection. Indeed because of the L5 rule, Luth will consider this intrusion attempt, $hp \rightarrow_{UDP} nt$ as an answer to the previous spoofed request, $nt \rightarrow_{UDP} hp$, even if in reality, $hp \rightarrow_{UDP} nt$ is a client datagram of the real intrusion attempt.

Due to its session establishment protocol, this scenario is up to our knowledge difficult to perform in TCP, that's why we authorized the equivalent rule in TCP (L3), even if the correct solution consists in both cases, developing the vulnerability monitoring MI-s[11].

The solution to this problem is to develop the vulnerabilities emulation MI associated to services using the UDP transport protocol. This is left for future work. We need now to investigate the 1620 remaining dropped TCP segments.

3.3.1.2 Dropped TCP segments

To investigate the reasons behind TCP segment drops, we extract the sessions containing dropped packets by means of the *tcpdump* utility. We then group such dumps by their size, assuming that if they have the same size, the patterns of the traces are similar and therefore the causes of drop decisions too[12]. The results obtained by the grouping of TCP sessions are depicted in Figure 3.13.

[11] Moreover, the L3 rule can be criticized too. We introduce the notation $cl \rightarrow_{TCP} sr$ to say there is a TCP session between the client cl and the server sr. After performing a successful and **non emulated** intrusion on the honeypot software, an attacker could use a communication channel like $attacker \rightarrow_{TCP} hp$ to take the control of the honeypot server, whether by using a backdoor, a "bindshell shellcode", or whatever name has been given to such a tool. Indeed L3 authorizes $attacker \rightarrow_{TCP} hp$ communications. However, the need for such a rule is motivated by the **emulation** of the same exact scenario by the honeypot, where the latter deliberately emulates the establishment of the previously mentioned communication channels. Let us say the attacker does not send a request to download a malware, i.e $< DownloadMalwareInInternet$ in $attacker \rightarrow_{TCP}$ channel, but a request to download a file present in the file system of the honeypot $< DownloadFilePresentInHoneypot >$. Both of the requests are accepted by the L3 rule. In the emulated case, the backdoor emulated by the honeypot is deliberately limited to the downloads of malware. Therefore, when receiving the $< DownloadMalwareInInternet >$ request the honeypot downloads the malware. But, when receiving $< DownloadFilePresentInHoneypot >$ the honeypot does not execute it and therefore, the file in the honeypot is not sent. However, in the real attack scenario, both requests are fulfilled by the real backdoor. Taking into account that this section does consider the real attack scenario as possible, Luth should drop the *DownloadFilePresentInHoneypot* requests while accepting $< DownloadMalwareInInternet >$. Even if not implemented yet, the developing of different vulnerability emulator's MI provides a solution to this problem.

[12] This approximation deserves its noun, indeed it is likely that two session dumps having the same size present different behaviours. However we consider that the benefits of an exact analysis of all the causes of the drop decisions, which requires quite a considerable effort, will permit to gain a negligible percentage on accepted packets.

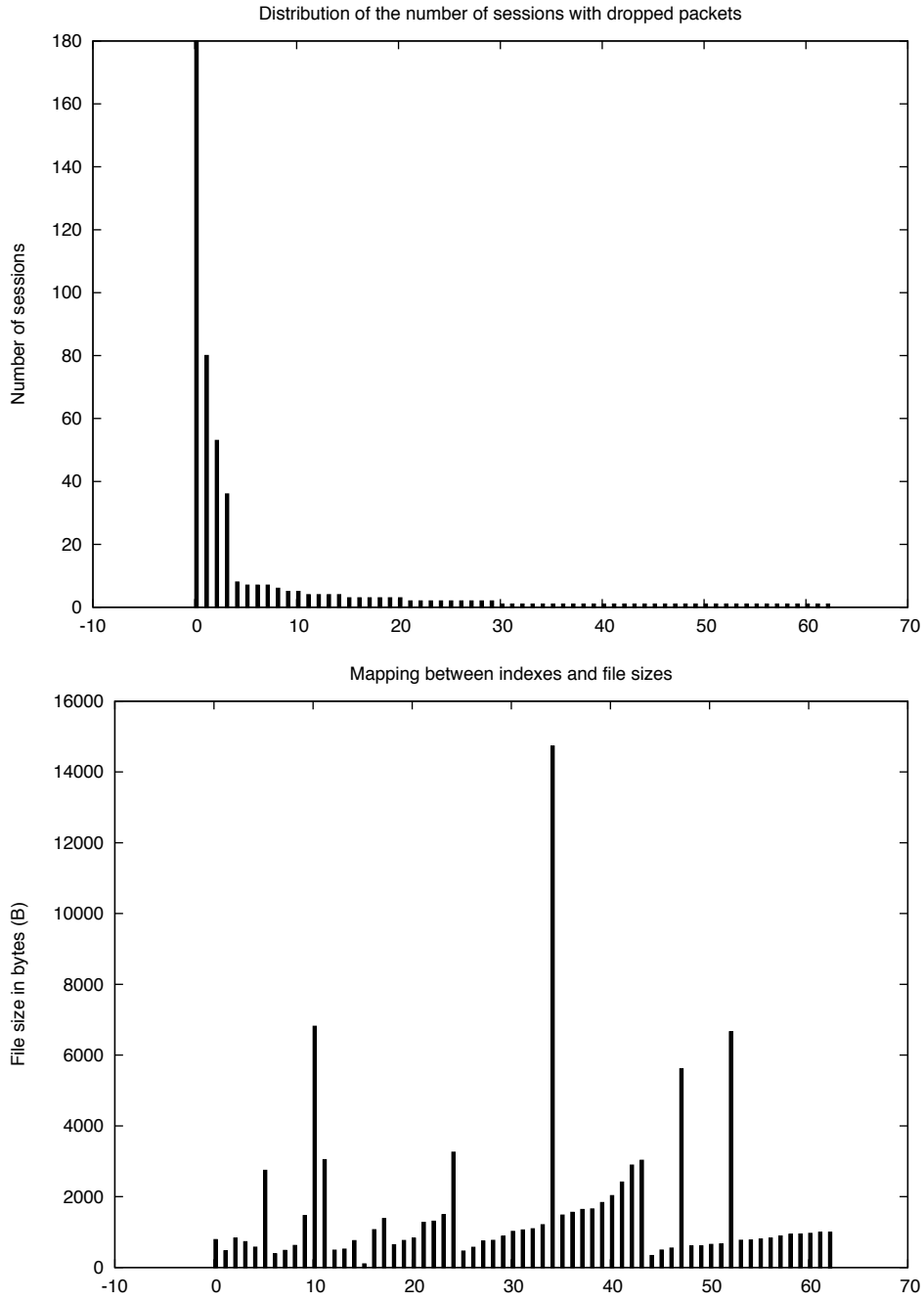


Figure 3.13: Characterization of TCP sessions containing dropped packets. Both graphics are related by the index represented in each abscissa. For example as shown by the plots having the 0 abscissa, there are 180 sessions grouped into a dump of about 700B.

The packets dropped in the four TCP sessions carrying more than 4000 bytes are the following (we consider them in decreasing order):

1. The sending of 60 RST after having closed an established a session targeting the emulation of Microsoft's SMB related vulnerabilities,

2. The sending of a RST segment after the honeypot sends a FIN to end an established session targeting the same emulator (twice),

3. The sending of 5 FINs from the malware, and successive FIN_ACK, that are then reset by the honeypot.

The other packets are dropped for similar reasons we group this way:

1. The sending of more than 1 retransmitted message inspected for session establishments (SYN,SYN_ACK, etc). Indeed, the computing of the actual midpoint automata with 1 retransmission takes a week in the server we have used. We have not even tried to launch the computing with 2 retransmissions, which is left for future work.

2. The sending of a RST by the malicious client after receiving a FIN from the server. Indeed as depicted in the Figure 3.11, the model of the malicious client does not take this behaviour into account. We have not be willing to model it in a new version of the automata because this discussable trend represents very isolated cases (5 packets among 88116).

3. The fine tuning of state related timeouts in TCP sessions. As depicted in [214], the timeouts represent a central role in the behaviour of TCP related sessions' inspection. Up to our knowledge, there is however no justified solution for managing the timeouts related to midpoint inspections. Nowadays, each inspection devices develop their own ad-hoc solutions. We have in our case, distinguished the timeout related to the messages to come in established session, from the messages managing sessions about to establish, or to close. Using this solution, the dropped packets are due to the fact that some sessions considered as ended from endpoints, are not from the midpoint point of views, and therefore a packet from a new session is considered as related to previous sessions, and vice-versa. We have not proposed solutions to this quite undeterministic problem.

To sum up the TCP case, all the drops are due to the monitoring of TCP session establishment messages, and do not involve the monitoring of downloads.

The other dump analyzed represents the same honeypot software activity during 5 weeks on another server in our laboratory. With the same rule, the inspection of 564209 packets reports the dropping of 15659 packets (2.78 %). The following section benchmarks the inspection performed when analyzing this dump.

3.3.2 Benchmarking the Inspection

To have an estimation of the side effects related to Luth performances, we compare the time to perform the computations by a process in userspace with the inter-arrival time of the packets, computed using the time-stamps present in the traffic dump. Indeed, if one Luth inspection does not finish before the next packet arrives, the packet will be buffered in a queue. The size of such a queue is limited. Therefore, if Luth is too slow some packets can be dropped without being inspected.

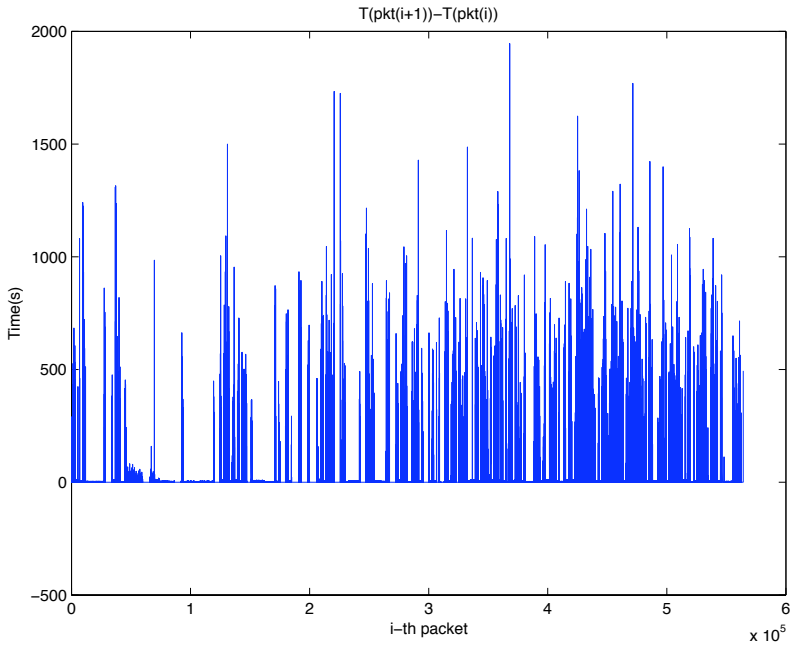


Figure 3.14: Packets inter-arrival times. A point (n,y) in the plot means that the n+1-th packet in the dump arrived y seconds after the n-th.

As depicted in the Figure 3.14, the inter-arrival time of packets going through the honeypot is not very demanding. Quite a lot of point in the plots show that after receiving or sending a packet, 500 seconds elapses before a new packet arrives or is sent. Indeed, the overall time needed to make the computations asked by Luth's inspection requires less time than packets to arrive. As depicted in Figure 3.15 in this global view, we don't see any point where Luth does not finish processing a packet before another one arrives. We identify as:

- $T(pkt(i))$ the time-stamp of the i^{th} packet in the dump,
- $D(Luth(pkt(i)))$ the duration of the inspection performed by Luth on the i^{th} packet.

To have a better understanding on what happens when the packet inter-arrival time is faster, like for example in malware downloads, we zoom the previous figure to the cases where $|[T(pkt(i+1))-T(pkt(i))]-D(Luth(pkt(i)))| < 5*10^{-3}$ as depicted in Figure 3.16.

We see that in some cases Luth is slower than the packet inter-arrival time, which will induce a bufferization of packets, and even packet drops if one of the involved buffer is full. To estimate the required buffer size needed so that packets are not dropped because of the buffer size limitations, we conduct the following experiment.

We estimate the evolution of the size of the buffer by emulating packets insertion and removal during the different inspections. We assume that all the processed packets are inserted in the buffer, considering for example that the buffer is oversized. We consider a packet is removed from the buffer once Luth finishes inspecting it. Let us say the buffer is empty and that a packet arrives at $T(pkt(i))$. Luth will finish processing the packet at $D(Luth(pkt(i))+T(pkt(i))$. The time at which $Luth$ finishes processing $pkt(i+1)$, $Tout(pkt(i+1))$, depends of the relation between $Tout(pkt(i))$ and $T(pkt(i+1))$. If $pkt(i)$

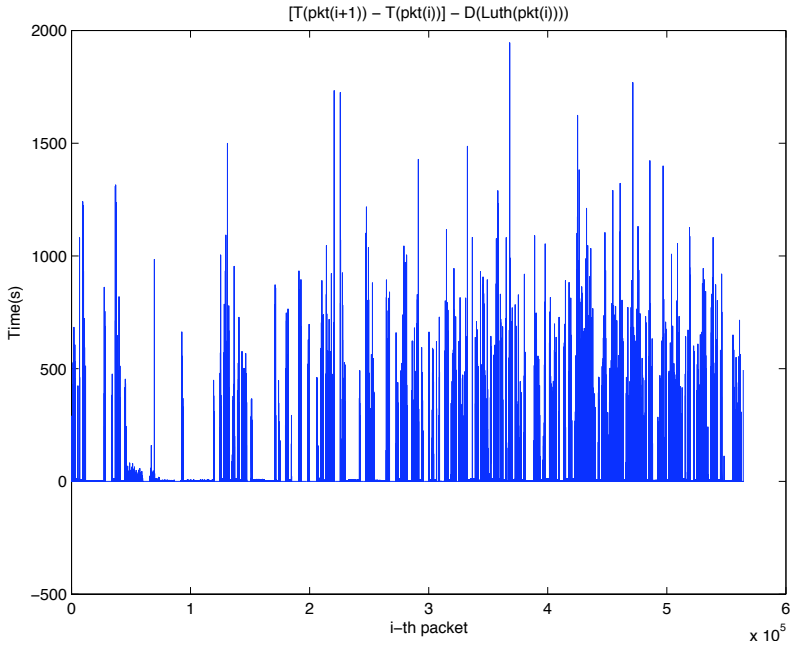


Figure 3.15: Estimation of the possible slow-down induced by using Luth. A point (n,y), with y<0 means that the n+1 packet comes faster than Luth processes the n-th packet.

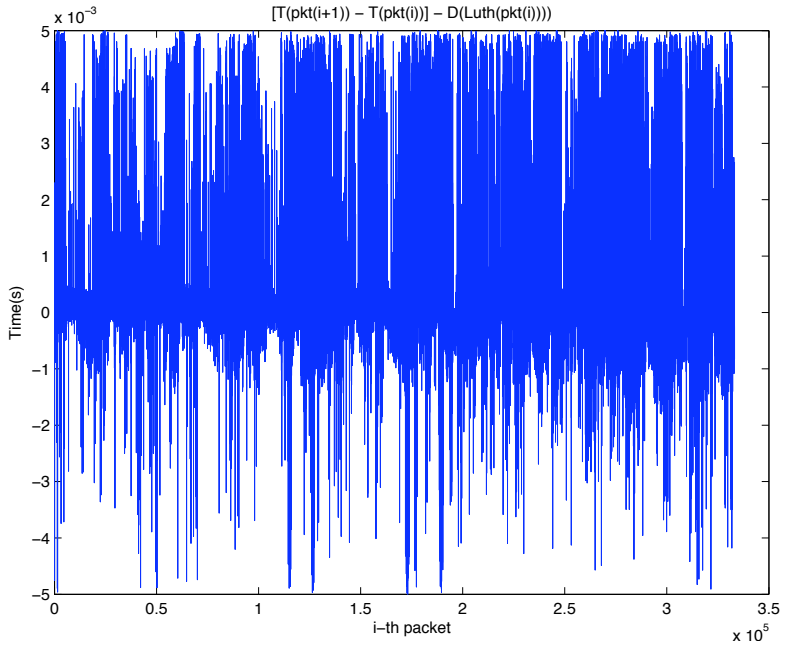


Figure 3.16: Zoom of the estimation of the possible slow-down induced by using Luth. A point (n,y), with y<0 means that the n+1 packet comes faster than Luth processes the n-th packet.

has already been processed at $T(pkt(i+1))$, i.e $Tout(pkt(i)) < T(pkt(i+1))$, then Luth will start processing $pkt(i+1)$ as soon as it arrives. Therefore $Tout(pkt(i+1)) = D(Luth(pkt(i+1))) + T(pkt(i+1))$. However, if Luth is still inspecting $pkt(i)$ at $T(pkt(i+1))$, i.e $Tout(pkt(i)) > T(pkt(i+1))$, then Luth will start inspecting $pkt(i+1)$ once the inspection of $pkt(i)$ is finished. In this case, $Tout(pkt(i+1)) = D(Luth(pkt(i+1))) + Tout(pkt(i))$. We group both formulas into $Tout(pkt(i+1)) = max(Tout(pkt(i)), T(pkt(i+1))) + D(Luth(pkt(i+1)))$. Finally, the number of packets present in a buffer at time $T(pkt(i))$ can be measured by counting the number of packets that will be removed after $T(pkt(i))$, i.e $|\{pkt(j), Tout(pkt(j)) > T(pkt(i)), j \leq i\}|$.

By using the previous formulas, the emulation of the insertion and removing in the buffer permits us to measure the number of packets in the buffer when processing $T(pkt(i))$ as depicted in Figure 3.17.

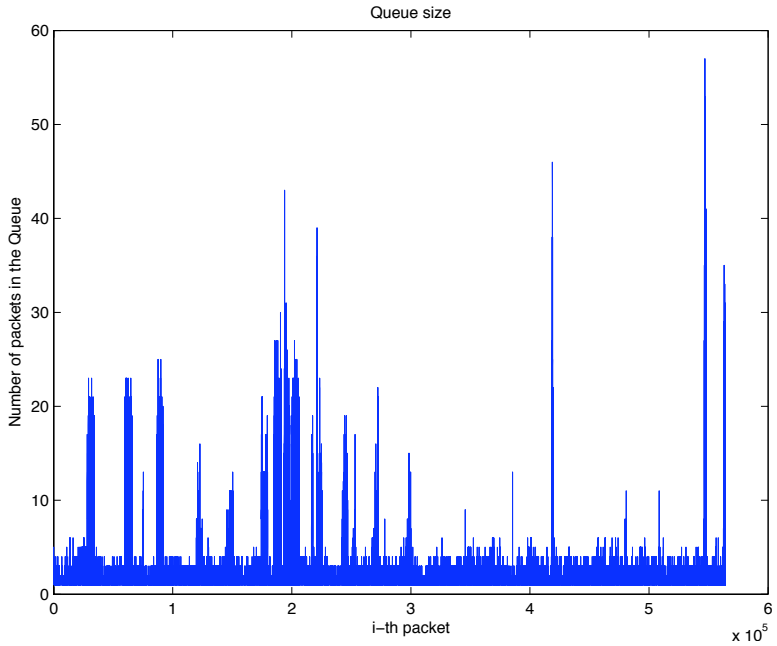


Figure 3.17: Evolution of the queue size before starting to inspecting a new packet. A point (x,y) should be read as: just before inspecting the x-th packet, there are y packets in the queue.

In the emulation of the experiment, we see that at most 60 packets are present in the buffer. Therefore by using one the function the available in the libnetfilter_queue_library to configure a buffer size of more than 60 packets during an online analysis presenting similar packets to analyze, packets may not be dropped because of the limitations of the queue size.

3.4 Conclusion

In this chapter, we have presented the problems that can be rose by an unmonitored malware downloading honeypot. Indeed, among others, the latter can be used as an intrusion proxy-server. We have then presented the MI we developed to monitor its network behaviours, while still letting it emulate vulnerabilities. The offline analysis shows that LUTH drops less than 3% of the packets related to the malware activity that reached our honeypots during 6 weeks.

To reduce the number of dropped packets, computing the TCP and MTCP midpoint automatas with more than one retransmissions can be explored (Section 3.2.1) . Regarding the dropped packets due to the tuning of timeouts, we are less optimistic (Section 3.3.1.2). Indeed, we do think that the solutions adapted to one context, will poorly behave on others.

We present in the last chapter, a method to analyze the malware collected thanks to such honeypots.

4

Dynamic Analysis of Malware's Network Communications

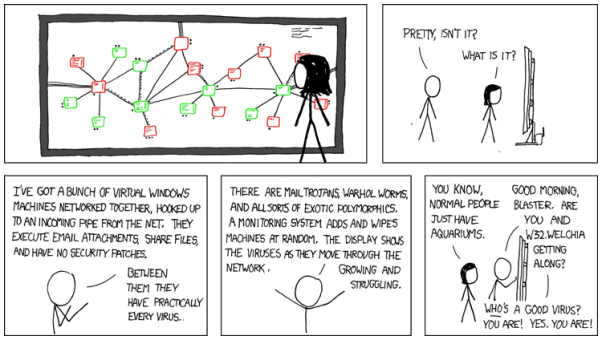


Figure 4.1: Extract of the webcomic xkcd, http://xkcd.com/350

This last chapter deals with the dynamic analysis of malware's network communication. First, the generic principles of this analysis are depicted, among others by describing the new MI-s we developed. Then, we describe the results we obtain by analyzing 31 different malware gathered using malware downloading honeypots.

4.1 Principles of the Dynamic Analysis

The need for monitoring dynamically analyzed malware's network communication towards the Internet has been explained in Section 1.2.3.2. Once the malware is executed in the sandbox and starts talking to a server in the Internet, we differentiate different ways of monitoring it. A first solution consists in letting the malware talk to the Internet and drop packets considered as unsafe. This solution presents two drawbacks. First, the intrusion attempts from the malware cannot be obtained from the experiment[1]. Second, as depicted in Section 2.1.1.2, the information analyzed by an MI is not always carried in a single message. When the information is fragmented, the sequence of already observed packets does not always give sufficient information to take a decision. Therefore, in these situations accepting a packet involves undeterministic consequences. As a solution to this problem, Vutukuru et al. propose in [234] to create a new decision, "wait", that buffers the packet until the necessary ones come. For very pragmatic reasons i.e.:

[1] If they are not dropped, the experiment will be considered as illegal.

- the *libnetfilter_queue* library (Section 2.3.2) does not give the opportunity to take such a decision, and,
- this library is the only available one to develop online userspace packet inspection,

we did not explore this solution. Instead, we perform such monitoring and analysis using an hybrid and recursive approach. As depicted in Figure 4.2, we split the network communications of a malware into three different groups, targeting:

1. Unkown services, labeled "To identify".
2. Services we emulate for safety reasons. These communications, like for example intrusion attempts, are labeled "To redirect in Lab".
3. Real services, labeled "To send in the Internet".

The recursive approach aims at making the set of communications labeled "To identify" empty. To do so, during an iteration of the experiment, a communication to identify is removed from the "To identify" and appended to the "To send in the Internet", or "To redirect in Lab" sets. This is achieved by redirecting, in a first step, unknown services towards a honeypot in our lab. These redirected communications are then analyzed by Luth. Once the identification is performed, the latter service is removed from "To identify" group and put in the set of communications "To send in the Internet", or in the set of the ones "To redirect in Lab". The experiment is then stopped and rebooted with the set of previously identified communications updated with the latest discovery. A new reboot is performed upon the apparition of a new unknown service, etc. Before implementing this recursive approach, we need to find a way to redirect the packets belonging to "To redirect in Lab" and "To identify" groups.

4.1.1 The Need for NAT

In the *IPv*4 Internet, the routing, i.e the process of carrying a packet to its correct destination, is performed according to the destination address present in the *IPv*4 packets. There are two kinds of *IPv*4 addresses [33]:

public: The ones known by any router in the Internet,

private: The ones only known by local routers.

Without considering multicast, anycast and broadcast protocols, a server receiving a packet with the source address *sIP*, answers using a packet with destination address *sIP*. If *sIP* is a public address, the packet sent to the Internet will come back to the initial sender. If it is private, the Internet routers do not know *sIP*. Therefore if the server is not local, at least one router will not be able to route the answer packet. When a router or more generally an interconnection device modifies the destination or the source[2] of one packet, it modifies its receiver or, the receiver of its answer. This feature has been used, among others, to cope with the lack of public *IPv*4 addresses in the Internet. Indeed, if a person or group of persons owns 10 computers with a single public *IPv*4 address, it cannot give Internet connectivity to the 10 computers without changing the addresses of the packets. The Network Address Translator-s (NAT) propose a way to do so [39]. In this case private *IPv*4 addresses can be given to 9 computers and

[2]Or both.

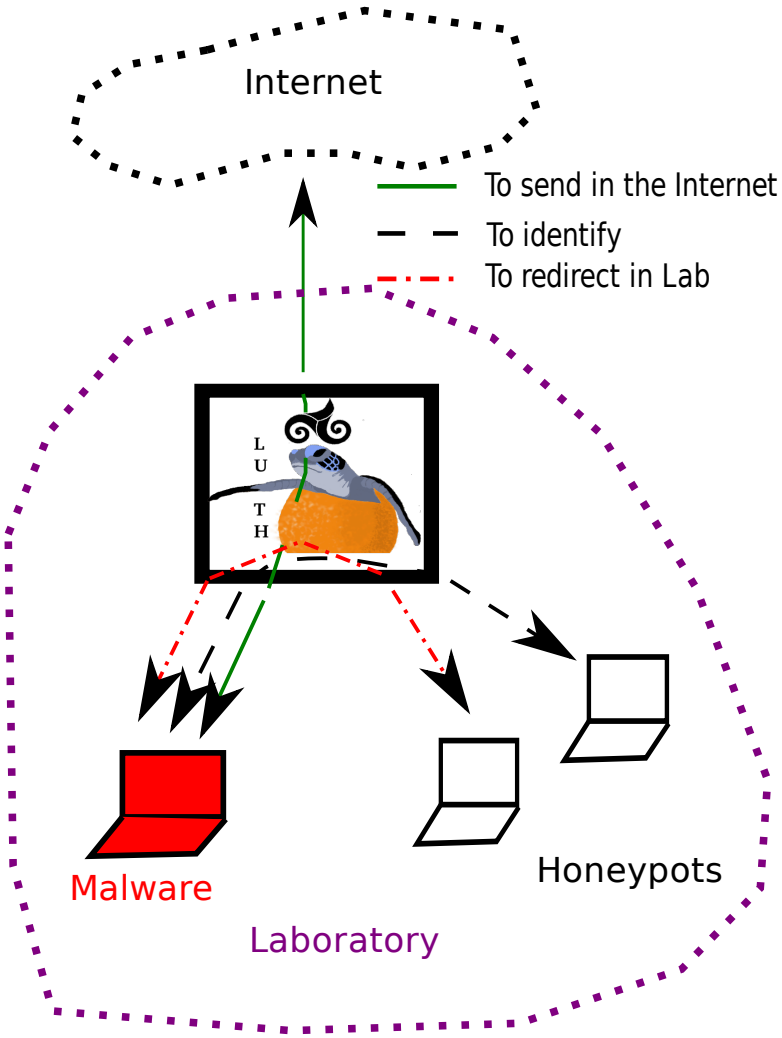


Figure 4.2: Dynamic monitoring of malware communications.

the public to the last one, which will act as a NAT router. The latter then makes the needed translations so that the public address is shared among the 9 computers. To redirect communications labeled "To identify" and "To redirect in Lab" we developed the NAT components to make LUTH behave like a NAT router as depicted in [60].

Problem 1 *Once the problem of redirections solved, three are remaining:*

1. *how to classify unknown services,*
2. *how to record the already classified services,*
3. *how to automate as safely as wanted the experiment.*

4.1.2 The Rule to Perform the Dynamic Analysis

These problems are resolved making quite an extensive use of Luth's functionalities. We do think the best way to understand the idea is to see the rule we use to dynamically analyze malware depicted in Listing 4.1. This rule is parameterized by five addresses:

1. *mw_ip*: private, the computer executing the malware,
2. *lab_net*: private, the net of the computers inside our lab,
3. *fk_ip*: private, the address of the honeypot[3],
4. *gw_ip*: public, the address of the gateway in the network interconnected to the Internet,
5. *lab_gw_ip*: private, the address of the gateway in the network interconnected to the lab network.

There are five different NAT rules:

NR1: Redirect the queries of the malware towards services using TCP on destination ports 445, 139 and 135 to the honeypot. The honeypot will bind the respective vulnerability emulators of the nepenthes honeypot in these ports. We chose to make this static decision for two reasons:

1. The malware collected with the nepenthes honeypot use most of the time vulnerabilities targeting services bound on these ports.
2. We have not developed vulnerability identifiers and monitoring MIs. Therefore we can only statically make this distinction for now.

The $max\_simul = 10$ parameter says that at most 10 simultaneous redirections are performed. The $rand = yes$ parameter implements a probabilistic redirection paradigm so that the result of the scanning of a network does not only detect the first 10 addresses. Indeed, this result is very unlikely in the Internet.

NR2: Redirect all the other ports towards a specifically developed server over TCP, bound in port 40000, that just reads all the bytes sent without answering anything. This aims at letting the malware send the first messages over TCP. Then, by analyzing thes messages, Luth's application identifiers MI over TCP identify the protocol over TCP used in this communication. The $max\_simul = 30$ states that more than 30 simultaneous redirections are not allowed.

[3]We have in a first step, used a single server to reduce the computing costs. However the approach can be generalized to multiple servers as depicted in Figure 4.2.

NR3: Redirect the remaining sessions to a "pit" which does not answer anything.

NR4: Redirect the communications of the honeypot ($Vm2$ in Figure 4.3) targeting the gateway server (gw_ip in Figure 4.3) to the malware ($Vm1$ in Figure 4.3). This trend is due to situations where once the malware detects it has a private address (mw_ip) natted by a public address (gw_ip)[4], and wants to make an attacked computer download a new instance of the malware from mw_ip. Let us say the malware performed a successful intrusion in a host in the Internet (here emulated by $Vm2$). To do so, as mw_ip is private, it is useless to send its own mw_ip address as the destination address of the post intrusion download query (here performed by $Vm2$). Instead, the latter sends the public address (gw_ip) hoping that in this situation, the NAT router ($Luth$) owning the public address (gw_ip) will translate the download attempt (towards gw_ip, and here from hp_ip) and redirect it to the malware's private address (mw_ip).

NR5: Give internet access to the download tries of the Nepenthes honeypot.

Listing 4.1: Monitoring malware

```
NAT(OIpv4(from=mw_ip,to=not (lab_net/24 or gw_ip))/
    OTcp(sr_port=445 or 139 or 135,rand=yes,max_simul=10) ->
    NIpv4(dst=fk_ip)/NTcp (* NR1 *) |
    OIpv4(from=mw_ip,to=not (lab_net/24 or gw_ip))/
    OTcp(sr_port=not (445 or 139 or 135),max_simul=30) ->
    NIpv4(dst=fk_ip)/NTcp(dport=40000) (* NR2 *) |
    OIpv4(from=mw_ip,to=not (lab_net/24 or gw_ip))/OTcp ->
    NIpv4(dst=fk_ip)/NTcp(dport=40001) (* NR3 *)|
    OIpv4(from=fk_ip,to=gw_ip)/OTcp ->
    NIpv4(dst=mw_ip)/NTcp (* NR4 *)|
    OIpv4(from=fk_ip,to=not (lab_net/24 or gw_ip))/OTcp ->
    NIpv4(src=gw_ip)/NTcp (* NR5 *)
);;
INSP([Arp (* MI1 *)|
    L2N(..)/Spm(..) (*MI2*) /Ipv4/[
      MTcp/TPort(cl_addr=mw_ip,sr_addr=not(lab_net/24 or gw_ip),
                 sr_port=445 or 139 or 135)/Void |
      Tcp/[TPort(cl_addr=mw_ip,sr_addr=not(lab_net/24 or gw_ip),
                 sr_port=not (445 or 139 or 135))/[
                IrcId(..) (*MI3 *)|
                HttpId(..)
           ] |
          TPort(cl_addr=fk_ip,sr_addr=not(lab_net/24))/[
            CSend|Link|Http(check_host=nepenthes)]
      ] |
      Udp/UPort(cl_addr=(lab_net/24 && not lab_gt_ip),
                sr_addr=lab_gt_ip,sr_port=53)/Dns
 ]]);;
```

The rules inside the $NAT(..)$ parenthesis represent the NAT rules at the begining of the experiment. The reader should notice that all the malware communications do not, in a first time, go outside the Lab. Therefore the only way a malware has to attack a third party ADPS is to exploit a bug in Luth. We won't

[4]For example, by asking in its C&C the actual source address of the communication observed by one of the botnet's master.

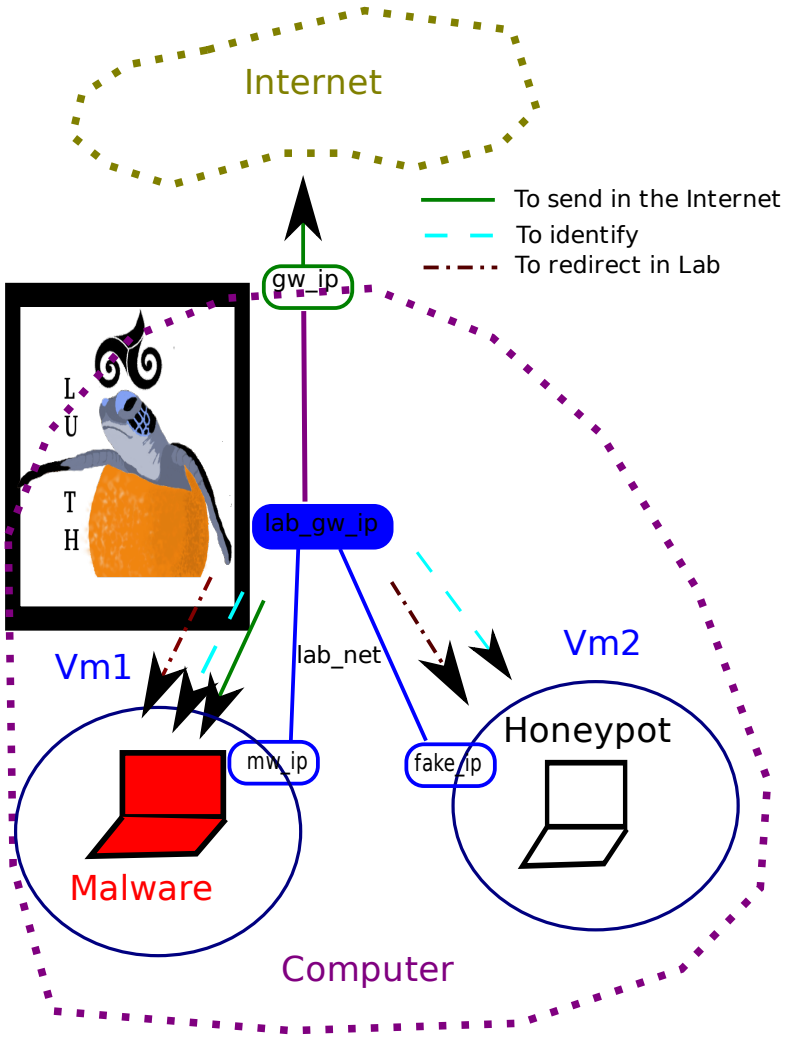


Figure 4.3: Platform to perform the dynamic monitoring of malware communications.

address here again the efforts done and the necessary remaining work to assume the absence of such bugs. Let us remind the three problems mentioned in Problem 1:

1. how to classify unknown services,
2. how to record the already classified services,
3. how to automate as safely as wanted the experiment.

We have here neither classified, nor recorded already classified services. However, to automate as safely as wanted the experiment, first, its first iteration should be considered as safe. By the NAT rules we have presented, during the first iteration of the experiment, no communications are sent to the Internet. That way, we prevent the possibility to attack a third party ADPS. Therefore, we consider the first iteration of the experiment as safe. The following section aims at showing how these problems are solved during the next iterations of the experiment.

4.1.3 Luth's Iterative Process to Analyze Network Communications

We describe in this section the details iterative process. First, we describe the *Arp* MI (MI1) ensures that a host can be identified by its source IPv4 address. The *Spm* MI (MI2) aims at identifying and blocking network attacks redirected in the Lab to prevent the flooding of the Lab network. These two MIs target are focused on the safety mentioned the third item of Problem 1. The last two MI, *IrcId* and *HttpId*, implement the classification and recording of unknown communications, while ensuring the safety of the next iteration of the experiment. In other words, they address the three problems of Problem 1.

4.1.3.1 Arp (MI1)

The inspection policy identifies network entities by means of their *IPv*4 addresses. Therefore, if for example the malware impersonates another address, it's activity won't be monitored anymore. Considering the interconnection technology we use in our experiment is *ETHERNET* (Section 4.3.1), the *ARP* protocol can be used to effectively impersonate other computers by means of "Arp cache poisoning" [53]. The *ARP* protocol is used to discover the mapping between *IPv*4 and *ETHERNET* mac addresses in an *ETHERNET* Local Area Network (LAN). The "Arp cache poisoning" works by sending malicious messages which pretend an attacked *IPv*4 address, *A*, belongs to the mac address of the computer, *M*, wanting to impersonate *A*. That way instead of *A*, *M* receives the messages with the destination address *A*. There are three hosts in the Lab LAN, and we know the *IPv*4 addresses belonging to the three hosts, together with their *ETHERNET MAC* address. Therefore the *Arp* MI is developed to drop *ARP* spoofing requests, using the known mapping between *IPv*4 and *MAC* addresses.

4.1.3.2 Spm (MI2)

The *Spm* MI is quite a rudimentary IDS detecting network attacks. It considers such an attack when the ratio between the number of *SYN* segments and the number of a set of packets exceeds the threshold given in the parameter of the MI [60] after having observed the number of packets given in the parameter *nb_pkt*. The packets are grouped into three sets in according to the target of a possible attack:

network scan : a network address at a single port,

port scan : a single address at a set of ports,

syn flood : a single address at a single port.

Network addresses are represented by means of CIDR notations:
$< ipv4\_address > / < network\_mask >$ [38].

We use boolean expressions (see *UPort* in [60]) on *TCP* ports to represent set of ports, like that:

```
type port
type 'a bool_expr = Val of 'a
      | Or of ('a bool_expr * 'a bool_expr)
      | And of ('a bool_expr) | Not of ('a bool_expr)
type port_bool_expr = port bool_expr
(* to illustrate how these boolean expressions are used we introduce
 * the following match_port_bool_expr and create_port functions:
 * val match_port_bool_expr bool_expr: port_bool_expr -> port -> bool
 * val create_port: int -> port
 * using these functions, we can express that
 * let twenty_two = create_port 22 in
 * let thirty_three = create_port 33 in
 * match_port_bool_expr (Or (Val twenty_two, Val thirty_three), twenty_two) = true,
 * match_port_bool_expr (Not (Val twenty_two), create_port twenty_two) = false
 *)
```

These expressions are compiled into a set of sorted intervals, reducing the evaluation of the boolean expression, to the checking of whether an address or port is inside one of the intervals. When matching a packet containing a *TCP SYN* segment targeting the address a on port p, the *Spm* creates three structures:

- $(a/24, p)$: an *IPv*4 network address and a port,
- $(a, Val\ p)$: an *IPv*4 address and a port boolean expression,
- (a, p) : an *IPv*4 adress and a port.

Each of these structures have moreover:

- a timeout, that is reinitialized each time a packet is considered to be related to the structure,
- the number of segment and *SYN*s incremented each time a segment is considered by the structures. Of course, the number of *SYN*s is only incremented if the segment has the flag 'S'.

When another segment $(a2, p2)$ is analyzed to the *Spm* MI, the structures are updated this way:

- $$\begin{cases} (a/24, p) \text{with incrementing the counters and updating the related timeout} & \Leftrightarrow a2 \in a/24 \\ (a/16, p) \text{idem} & \Leftrightarrow a2 \notin a/24 \wedge a2 \in a/16 \\ (a/8, p) \text{idem} & \Leftrightarrow a2 \notin a/16 \wedge a2 \in a/8 \\ (a/24, p) \text{ without incrementing the segment counters and updating the timeout} & \text{else} \end{cases}$$
- $$\begin{cases} (a, Val\ p) \text{with incrementing the counters and updating the related timeout} & \Leftrightarrow a2 = a \wedge p2 \in Val\ p \\ (a, (Val\ p)\ Or\ (Val\ p2)) \text{idem} & \Leftrightarrow a2 = a \wedge p2 \notin Val\ p \\ (a, Val\ p) \text{ without incrementing the segment counters and updating the timeout} & \text{else} \end{cases}$$

- $$\begin{cases} (a,p) \text{with incrementing the counters and updating the related timeout} & \Leftrightarrow a2 = a \wedge p2 = p \\ (a,p) \text{ without incrementing the segment counters and updating the timeout} & \text{else} \end{cases}$$

If the segment towards $(a2, p2)$ is a SYN, and all the previous computations go to the else cases, a new triple structure is created with (a, p). Finally once, the packet count of syn segments divided by the number of packets related to one structure is superior to the threshold the packet is dropped.

4.1.3.3 HttpId and IrcId (MI3)

The last MIs to describe are *HttpId* and *IrcId*, which aim at identifying *IRC* and *HTTP* protocols and **at implementing the recursive approach of our experiment**. We choose to only describe the functioning of the *IrcId* MI, as the *HttpId* is working similarly. The aim of *IrcId* is to identify among the communications labeled "To identify", the ones targeted to a server, parameterized by an address *irc_srv_ip* and TCP destination port *irc_srv_port*[5], which speaks a dialect of the *Irc* protocol identified by the *irct* parameter.

Once the server is identified (1. of Problem 1), i.e. this information is recorded (2. of Problem 1) by means of two added NAT and INSP rules that configure the safety of next iteration (3. of Problem 1).

These two rules are parameterized by the parameters of the *IrcId* depicted in Listing 4.2[6].

Listing 4.2: IrcId Parameter

```
IrcId(irct=ircs,public_ip=gw_ip,
      new_rule_layering=L2N(..)/Spm(..)/Ipv4(..)/
      Rt(max_bw=50kbs,from=previous_client,to=*)/
      Tcp(tm=24hr)/
      TPort(cl_addr=previous_client,sr_addr=previous_server,
      sr_port=previous_server)/Irc(irct=ircs))
```

The *public_ip* gives the public address used in the experiment. This address is used to compute the NAT rule to add for letting the non identified protocol speak to the Internet. The *new_rule_layering* argument describes **the layering to add in the INSP rule of the next iteration of the experiment**, once a service is identified as belonging to the protocol to identify. As depicted in [60], *max_bw* stands for maximum bandwith. Therefore, the written *RT* MI just ensures that no more than 50kbs are sent to the Internet. The *TPort* MI verifies that the malware talks to the identified servers address and ports. The last $Irc(irct = ircs)$ MI ensures that the service identified by the IrcId during a previous iteration, talks standard or RFC compliant [37], Irc. Indeed,

- *irct* stands for irc type, and
- *ircs* stands for irc standard.

However, another layering could be used to configure this appended rule. For example, a more permissive layering is depicted in Listing 4.3.

[5] A layering $Udp/../IrcId$ is not considered as valid by Luth's current implementation.

[6] The *HttpId* works similarly.

Listing 4.3: IrcId Other parameter

```
IrcId(irct=ircs,public_ip=gw_ip,
      new_rule_layering=L2N(..)/Ipv4(..)/
      Tcp(tm=24hr)/
      TPort(cl_addr=previous_client,sr_addr=previous_server,
      sr_port=previous_server)/Void
```

Indeed, thanks to the layerings in the new_rule_layering parameters, the user can configure as precisely as wanted the safety of the experiment.

The mechanism used to substitute the values of $previous_*$ variables here is similar to the computing of data channels in *FTP* MIs. However, to inform Luth that the experiment should be rebooted once the new layering is incorporated in the new rule, the *IrcId* MI's do not output *Add_child* decisions, but *New_rule* ones. The only difference is that once the layering computed by *analyse_res_*$*$ functions reaches the root of the inspection tree, the malware communication analyzer stops all the experiment and computes a new rule this way:

1. Appending a NAT rule to give internet connectivity for this newly identified service,
2. Appending the computed layering by means of the join function into the inspection tree.

For a more illustrative example, the rule computed once an $IRC(irct = ircs)$ server is identified can be seen in Listing 4.4:

Listing 4.4: Rule after first iteration

```
NAT(
  OIpv4(from=mw_ip,to=irc_srv_ip)/OTcp(dport=irc_srv_port) ->
  NIpv4(src=gw_ip)/NTcp |
  <..> (* previous_rule *)
);;

INSP(Arp |
   L2N(..)/Spm(..)/[
     (* start of appended MIs *)
     Ipv4(..)/[
       Rt(max_bw=50kbs,from=mw_ip,to=*)/Tcp(..)/
       TPort(cl_addr=mw_ip,sr_addr=irc_srv_ip,sr_port=irc_srv_port)/
       Irc(irct=ircs) (*NMI*) ]
     (* end of appended MIs *)
    | <..> (* previous rule *)
       ]
  ]
);;
```

By this mechanism, we propose a solution to the 1^{st} and 2^{nd} items of Problem 1. The third item addresses the safety of the experiment. The safety of the layering used to inspect the identified Irc communication going to the Internet (NMI) is open to debate. For example, the need for the *Spm* or the *RT* MI is subjective. Indeed, one could say they are useless, others could argue it's too dangerous to let a malware send 50kbs in the Internet. The reader should however remember that the newly computed layering (NMI) is factorized by means of the *join* algorithm (Section 2.2.2.5). Therefore, the malware will not be able to send $n * 50kbs, n$ being the number of identified $Irc(irct = ircs)$ communications, but

50*kbs* at most to all these identified servers. Following the "paranoid" paradigm, this layering does not among others, prevent the malware from performing naptha attacks (Section 1.1.2.2), even if it could be done using the *max_session* parameter of the Tcp MI[7]. These configurations result from our subjective point of views. However, we should take into account that the solutions considered as 'right' by the community working in this field of research, has, up to now, only been motivated by the observation of a malware that validates or contradicts one of the previous statements. Therefore, there are up to now no scientific justifications to argue in favor of a layering against another one, and our contribution does not solve this problem. However, for the reasons we mentioned in Chapter 2, Luth is flexible enough to express all possible subjective point of views:

From the most paranoid: which will have less information but which will hardly attack a third party ADPS.

To the most adventurous: which will have greater probability of getting more information and more attacks towards third party ADPS.

We can now justify that we effectively propose a solution to the last item of Problem 1: "how to automate as safely as wanted the experiment.", focusing our contribution to the implementation of the "as safely as wanted" words.

4.2 Implementation

The main work left to do is divided between the implementation of Network Address Translation, and the tuning of decisions and "reduce" function. The implementation of Network Address Translation has been done in userspace, without relying on the NAT functionalities provided by the SNAT, DNAT and connmark features of the Netfilter framework for the reasons depicted in Section 4.2.1.1. Even if quite challenging, the complete implementation of Nat functionalities in userspace is possible thanks to the Netfilter functionalities depicted in Section 4.2.1.2[8]. Finally, as we won't introduce more decisions in the **DS** set in this Ph.D, we finally present the "reduce" and related **DS** decisions (Section 2.2.2.2) in Section 4.2.2.

4.2.1 Network Address Translation

This section first addresses the reasons why we fully implement Nat functionalities in userspace. Then, after performing an overview of Netfilter, we present how these functionalities are implemented and integrated to the "inspect" function of Luth.

4.2.1.1 The Need for Userspace NAT

The Nat functionalities offered by the SNAT, DNAT and connmark features of the Nat router present in the GNU/Linux operating system do not implement the random redirections needed by the NR1 rule of Listing 4.1. The implementation of this feature has motivated the work done in [56]. This solution works by dynamically editing the configuration of already available DNAT and connmark features. It presents however three drawbacks:

[7] We choose not to take this problem into account because we do think the job done by an IRC server is light enough, unlike the job done by an SSH or HTTP server, so that it can handle considerable amount of sessions at the same time.

[8] We'd like here take the time to thank the developpers of Netfilter for developing such an extensible framework.

1. It's developed in an unsafe way (non proved C code),

2. It presents the same limitations as the other midpoint inspection devices. Indeed, it can be configured to translate the parameters of a set of *monolithic protocol layerings*. Even if we have not taken the time to study a a nat function similar to our inspect function, this work seems promising. Here again, we plan to implement such a prototype in Luth which is in userspace.

3. It's difficult to integrate it with the "inspect" algorithm.

For these reasons, we develop a basic Network Address Translator using *monolitic protocol layerings* and providing random redirections using a statically type-safe language, in userspace. The problems we have identified and solved in the following sections should therefore be seen as the first steps towards the implementation of a non monolithic nat router interacting with the "inspect" algorithm of a midpoint inspection device. This implementation has been done using the features of Netfilter we describe in the following section.

4.2.1.2 Netfilter Overview

As depicted in Section 4.1.1 performing network address translations modifies the routing of a packet. Netfilter, GNU/Linux's framework to implement firewall and NAT router, is a set of different **tables**, inserted in the different **chains** traversed by a packet when being processed by this operating system. The interactions between the tables, the chains and the routing decisions are depicted in Figure 4.4. Netfilter has five built-in[9] chains:

PREROUTING : Labeled PRE, which analyzes packets before they are routed.

INPUT : Labeled INPUT, which analyzes packets addressed to the computer executing Netfilter, before reaching the respective network programs.

FORWARD : Labeled FWD, which analyzes packets that go through the computer executing Netfilter.

OUTPUT : Labeled OUT, which analyzes packets having a source address belonging to one of the interfaces of the computer executing netfilter.

POSTROUTING : Labeled POST, which analyzes packets going in the Internet, after they have been routed.

The NAT is configured in the Snat and Dnat tables. Snat stands for Source natting, and Dnat for destination natting. Said otherwise, the NAT configured in the Snat table will translate the source address[10] of a client and, as a consequence, the destination of its server answers. The NAT configured in the Dnat table does the analogous process. One should notice that unlike for example the mangle table, the Dnat is only present in PREROUTING and OUTPUT chains. Similarly, the Snat is only present in the POSTROUTING chain[11].

[9] A user has the possibility to add new chains.

[10] And eventually ports.

[11] The reason behind the position of the Dnat chain is that it must be before the routing process. Indeed, changing the destination address of a packet changes the routing of the packet. One of the reasons why Snat might have been put in POSTROUTING is to prevent mistaking packets sent by the host from the ones Snatted by the host in the OUTPUT chain.

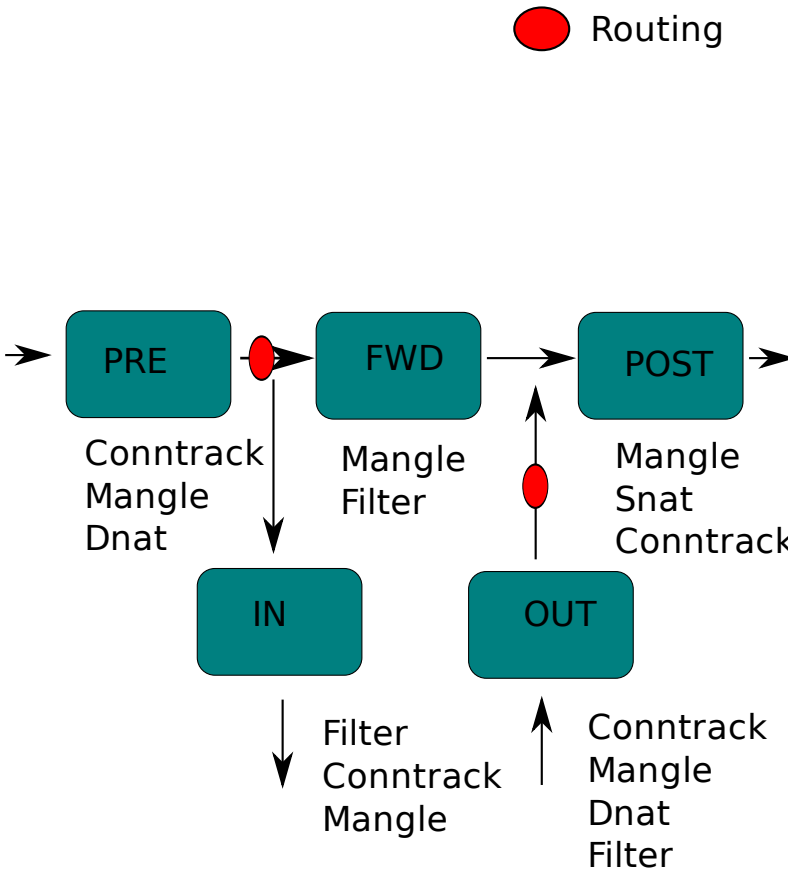


Figure 4.4: Netfilter, iptables and routing in GNU/Linux.

The configuration of netfilter is done by means of the "iptables" utility [26]. This utility configures what decisions should be taken concerning selected packets. The selection criterion involves a predefined set of packets properties together with its associated set of automatically or manually updated state variables. Netfilter gives the possibility to take five different decisions [19]:

NF_ACCEPT : continue traversal as normal.

NF_DROP : drop the packet; don't continue traversal.

NF_STOLEN : I've taken over the packet; don't continue traversal.

NF_QUEUE : queue the packet (usually for userspace handling).

NF_REPEAT : call this hook again.

The NF_QUEUE decision enables, as said in Section 2.3.2, to select the packets we want netfilter to send to our process in userspace. The communication between the userspace program and netfilter is achieved by means of the libnetfilter_queue library. This library gives the opportunity to communicate a verdict by using the two functions depicted in Listing 4.5.

Listing 4.5: Verdict API

```
int nfq_set_verdict(
 struct nfq_q_handle *qh, u_int32_t id, u_int32_t verdict,
 u_int32_t data_len, unsigned char *buf);
/* qh: file descriptor to communicate with netfilter.
   id: packet identifier.
   verdict: the decision to take.
   buf: pointer to the modifed packet
        (NULL means the packet is not modified).
   data_len: length of the modified packet
             (With NULL the length should be 0).
*/

int nfq_set_verdict_mark(
 struct nfq_q_handle *qh, u_int32_t id, u_int32_t verdict,
 u_int32_t mark,u_int32_t data_len, unsigned char *buf);

/* same as set_verdict only the mark mark is given to the packet with
   id id. BEWARE: the mark is the one matched by the ``mark'' state
   variable handled by the ``mark'' module, not the ``mark'' state
   variable handled by the ``connmark'' module.
*/
```

There is no manual for this library, therefore the conclusions written in commentary are open to debate, as they result from personal conclusions made after experimenting the library. After having presented the notions necessary to implement the NAT in userspace, the next section deals with its actual implementation. This implementation addresses two different problems:

1. The Integration with Netfilter,
2. The Integration with the inspect function of Luth,

We address both problems in the following two sections.

4.2.1.3 Integrating Userspace NAT with Netfilter

Problem 2 *The implementation of userspace Nat needs first to interact with Netfilter to:*

1. *Obtain the packets to analyze, and*
2. *communicate the modified version of the packet to Netfilter,*
3. **at precise locations** *in the routing process. As depicted earlier the source address can only be changed in POSTROUTING chains, and the destination address in PREROUTING chains.*

There are three functionalities provided by Nefilter that permit to solve these problems in userspace:

1. The NF_QUEUE decision so that netfilter sends packet and wait answers, (1. of Problem 2)
2. The possibility to update a variable called "mark", associated to each packets. This feature is used to store in the packet a state used by LUTH. This will be used to send the packet to modify in appropriate locations (2. of Problem 2)
3. The nfq_set_verdict and nfq_set_verdict_mark decisions, to modify the mark, source and destination addresses of the packets. This will be used to edit and consult the previously mentioned mark (3. of Problem 2).

Finally, as we implement a full Nat router in userspace, we need to take into account **all** the packets involved in a communication. Among others, this means that during a SNAT-ed session, the packets from the client should have their source address changed. Similarly, server packets should have their destinations address changed. The analogous requirements are imposed by DNAT-ed sessions.

The INSP rule used in our dynamic analysis only accepts three different types of communications:

1. Redirected communication to honeypots: labeled "To identify" and "To redirect in Lab".
2. Communications sent to the Internet: labeled "To send in Internet".
3. Communication with the DNS server which is in this platform executed in the same host as Luth (which can be considered as "To send in Internet").

The rules depicted in Figure 4.5 and in Listing 4.6 exist for the following reasons:

- The client packets of the first type of communication must be Dnated in PREROUTING (PR.1),
- The server packets of the first type of communication must be Snated in POSTROUTING (PO.5),
- The client packets of the second type of communications must be Snated in POSTOURING (PO.4),
- The server packets of the second type of communications must be Dnated in PREROUTING (PR.2),
- The client packets of the third type of communications must be inspected. This can be done anywhere, but as they are already caught in PR.1, we don't add a new rule for that. (PR.1)

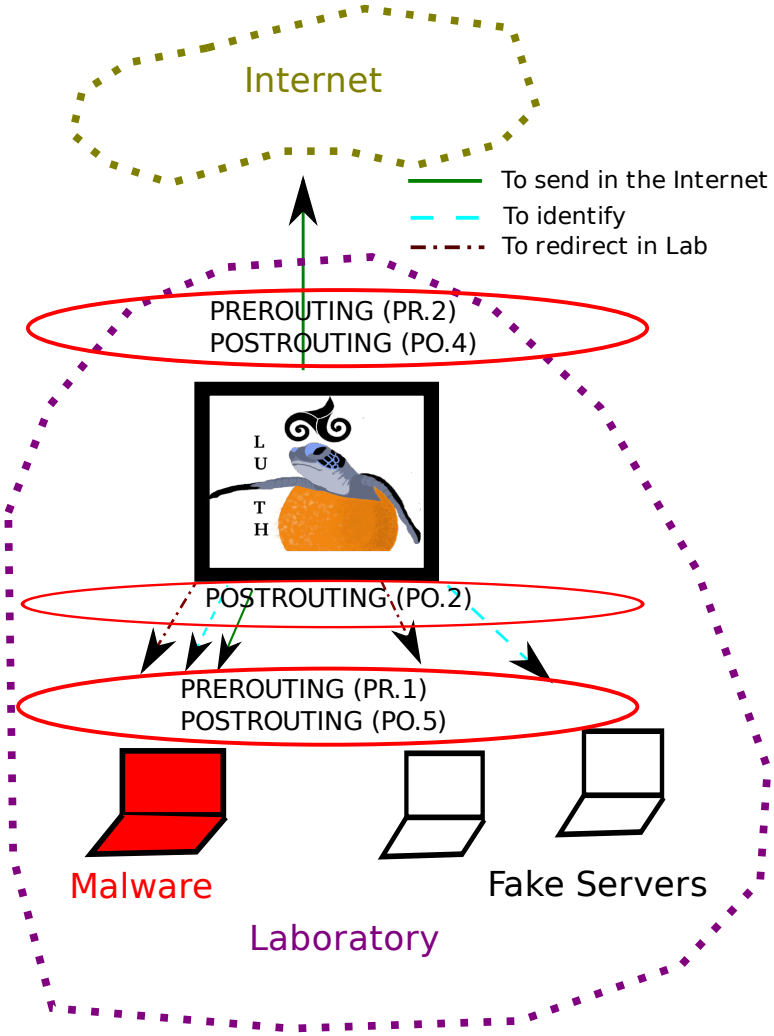


Figure 4.5: Illustration of the location of iptables rules.

- The server packets of the third type of communications will only be caught in POSTROUTING rules. Indeed, the DNS server is executed in the same host as Luth. Therefore, packets from the DNS server to the malware go through the path OUTPUT-POSTROUTING (Figure 4.4). We add another POSTROUTING rule (P0.2) to differentiate them from the packets that go through all the PREROUTING rules without having been inspected. Indeed, when all the buffer involved in the queues used to communicate with netfilter are full (Section 2.3.2), packets traverse the chains without being inspected.

Listing 4.6: Hooking netfilter for userspace natting

```
# PREROUTING rules

# PR.1 inspect packets coming from malware and honeypots
iptables -t mangle -A PREROUTING -m physdev --physdev-is-in\
-j NFQUEUE --queue-num 0

# PR.2 snatted packets going back from internet
iptables -t mangle -A PREROUTING -i $if\
-j NFQUEUE --queue-num 4

# POSTROUTING rules

# PO.1 Luth said we are done (packet dropped, accepted (DNS) or
# accepted, and Dnated)
iptables -t mangle -A POSTROUTING -m mark --mark 0x04/0xff\
 -j ACCEPT

# PO.2 answers from servers in the gateway (DNS)
iptables -t mangle -A POSTROUTING -s $lab_gt_ip\
 -d lab_net -m mark --mark 0x00/0xff -j NFQUEUE --queue-num 2

# PO.3 all remaining packets with mark 0 must be dropped
# (NFQUEUE buffers full)
iptables -t mangle -A POSTROUTING -m mark --mark 0x00/0xff -j DROP

# PO.4 snat of packets ``To send in Internet''
iptables -t mangle -A POSTROUTING -o $if -m physdev\
--physdev-is-in -j NFQUEUE --queue-num 1

# PO.5 snat of dnatted ``To redirect in Lab/To identify''
# packets going back
iptables -t mangle -A POSTROUTING -m physdev --physdev-is-out\
-j NFQUEUE --queue-num 3
```

The details of the implementations are quite stodgy and won't be addressed here. The curious readers are nevertheless invited to follow these descriptions in [58].

Finally, we should inform the reader that the Nat MIs have been developed taking into account the consideration depicted in the RFC 5382 [25] without implementing "hairpinning". "Hairpinning" involves Snating and Dnating the same packet. We don't implement this functionality for two reasons:

1. it is more complex than just Snating or Dnating a packet.

2. We don't need "hairpinning" to implement our malware's network communications dynamics analysis platform.

We have in this section described why and how we implemented the Nat MI described in [60]. The following section deals with the definition of the **DS** set and of the "reduce" function first introduced in Section 4.2.2.

4.2.2 Definition of the DS Set and of the Reduce Function

A last problem the platform needs to solve is to find a way to express in the Luth rule, so that already identified communications, are not identified again.

To have a better understanding of this problem let us take a closer look on the rule computed after the identification of an Irc server, as depicted in Listing 4.7, to illustrate a problem the definition of **DS** set and "reduce" function address.

Listing 4.7: Rule after first iteration

```
...
INSP(Arp |
   L2N(..)/Spm(..)/
      Ipv4(..)/[
       Rt(max_bw=50kbs,from=mw_ip,to=*)/Tcp(..)/
       TPort(cl_addr=mw_ip,sr_addr=irc_srv_ip,sr_port=irc_srv_port)/
       Irc(irct=ircs) ] (*L1*)
     (* end of appended MIs *)
    | Ipv4/[
      MTcp/TPort(cl_addr=mw_ip,sr_addr=not(lab_net/24 or gw_ip),
                 sr_port=445 or 139 or 135)/Void |
      Tcp/[TPort(cl_addr=mw_ip,sr_addr=not(lab_net/24 or gw_ip),
                 sr_port=not (445 or 139 or 135))/[
                   IrcId(irct=ircs,..) | (*L2*)
                   IrcId(irct=ircu,..) | (*L3*)
                   IrcId(irct=ircp,..) | (*L4*)
                   IrcId(irct=hallo,..) | (*L5*)
                   HttpId(..) ]

      ]
   | (* end of the rule *)
     ..
);;
```

Up to now, we will say that DS contains *Accept* and *New_rule* decisions. In the following lines we show that without adding another decision, already identified communications would indefinitely be identified at each iteration.

Let us say that a sequence of packet *newRuleSeq* is first analyzed by Luth. Then, the *sNewRule* segment towards the destination couple (irc_srv_ip,irc_srv_port) arrives. When analyzing this segment, the layerings L2,L3,L4,L5 (Listing 4.7) output the *New_rule* decision creating the L1 layering in Listing 4.7. Let us see the decisions taken by the L2,L3,L4 and L5 layerings more in detail. After inspecting *sNewRule*, L2 computes *New_rule*. Taking into account that *IRC* dialects are close, L3, L4 or L5 will consider that *sNewRule* does not give sufficient information to identify it

as being part of their dialect. However, *sNewRule* can be close enough to one dialect so that one these MI are willing the see other segments of the session, conducting to an eventual other *New_rule* decision. For this reason, let us say L3 outputs *Accept*. The computed decision by all the inspection tree, is therefore $reduce(\{New\_rule, Accept, ...\})$. So that L1 is computed, we must have $reduce(\{New\_rule, Accept, ...\}) = New\_rule$. The experiment is stopped, L1 is inserted, and the experiment is rebooted. We are now in the second iteration of the experiment. Let us say the malware sends again the same previous sequence of TCP segments, *newRuleSeq* and that *sNewRule* arrives. When analyzing the *sNewRule* segment, *L1* computes *Accept*. The decision computed by L2,L3,L4,L5 is $reduce(\{New\_rule, Accept, ...\}) = New\_rule$. Therefore, the decision taken by all the inspection tree will be $reduce(\{Accept, New\_rule ...\}) = New\_rule$. However, this rule will append the same L1 layering as before to the inspection tree, which will unnecessarily reboot the experiment all the times the malware sends the sequence *newRuleSeq* and then *sNewRule*.

We call this problem *New_rule* loop. To solve this problem we add the $Buffer$ decision to our decision set. The definition of the "reduce" function and **DS** set used by our prototype are depicted more in detail in Listing 4.8.

Listing 4.8: Ds and reduce

```
type ds =
 (* dynamic analyzer *)
   New_rule of (..)
 | Buffer
 (* filter*)
 | Accept
 | Drop
 | Add_child of ((..) set)
 (* Ids *)
 | FloodDrop

(* reduce function used by Luth *)
let reduce_couple x y =
  match (x,y) with
    | (FloodDrop,_) -> FloodDrop
    | (_,FloodDrop) -> FloodDrop
    | (Add_child xl,Add_child r) -> Add_child (union xl r)
    | (Add_child _,_) -> x
    | (_,Add_child _) -> y
    | (Accept,_) | (_,Accept) -> Accept
    (* need to do something to prevent this rule *)
    | (New_rule _, New_rule _) -> assert(false)
    | (New_rule _,_) -> x
    | (_,New_rule _) -> y
    | (Buffer,_) | (_,Buffer) -> Buffer
    | _ -> Drop

let rec rec_reduce curr_ds ds_set =
  if is_empty ds_set then curr_ds
  else
    let one_ds = choose ds_set in
      rec_reduce (reduce_couple curr_ds one_ds) (remove ds_set one_ds)
```

```
let reduce dsset = rec_reduce Drop ds_set
```

First, we present how these definitions validate the property defined in Definition (Section 2.2.2.5). Then, we show how these definitions solve the *New_rule* loop problem.

The property the "reduce" function needs to verify is $\forall X \in \wp(\wp(\mathbf{DS})), reduce(\{reduce(x), x \in X\}) = reduce(\bigcup_{x \in X} x)$ Taking apart *Add_child* decisions, and simultaneous *New_rule* decisions, reduce_couple implements a relational order in the decision set. Therefore, reduce can be seen as choosing the maximum element on the set. Taking *DS* the set of decisions without *Add_child* and *New_rule*, we have $\forall X \in \wp(\wp(\mathbf{DS})), sup\{sup\, x, x \in X\} = sup\{\bigcup_{x \in X} x\}$, which validates the property defined in Definition (Section 2.2.2.5). If we consider a set where an *Add_child* decision appears with $FloodDrop$ decisions, the result will always be $FloodDrop$. Without $FloodDrop$, the decision will be in both cases the union of all the sets in the *Add_child* decisions, which validates the property too. Finally handling two *New_rule* decisions can be done by desiging the append operation on two rules. We have not yet investigated this possibility and that's why the assertion has been written. This study is left for future work too.

The solution to the *New_rule* loop problem resides in the $Buffer$[12], *New_rule* and *Accept* rules. The *Accept* decision has priority over *New_rule*. Therefore once L1 answers *Accept* and L2 answers *New_rule*, the reduce function takes the wanted *Accept* decision. The $Buffer$ decision aims at not disturbing the L2,L4,L5 decisions. These MI takes the $Buffer$ decision to express they are waiting information to perform protocol identification. This decision has lower priority than *New_rule* and therefore does not prevent other MI-s to create the computing of *New_rule*-s.

We have in this section detailed the main principles of the implementation of our dynamic analysis paradigm. The next section explains the experimentation performed on 31 malware samples.

4.3 Experimentation

After describing the platform where the presented software has been installed, this section describes the results obtained by the dynamic analysis of 31 malwares.

4.3.1 Description of the Platform

As depicted in Figure 4.6 the platform is implemented in a single computer using a single real network interface. The hosts executing the malware and the honeypot are implemented by virtual machines and are interconnected by means of a software bridge [14] and associated Tap virtual interfaces [46]. This interconnection technology has been chosen because it integrates very well with Netfilter, and therefore with Luth. We have used the Kernel-based Virtual Machine (KVM) [18] for three reasons:

- Possibility to interconnect them using Tap devices and the previously mentioned software bridge.
- Possibility to install Windows and Linux Operating System on the virtual machine. For example we did not manage to install our Windows image using Xen.
- The use of Intel-VT or AMD-V virtualization technology. Indeed, some malware sample stopped their execution on the similar Qemu [32] software, both in emulation and accelerated mode (using

[12]The reader should not mistake the decisions taken by Luth, with the decisions output to Netfilter. For example *New_rule* decisions generates a NF_DROP, and reboots all the experiment. Similarly a $Buffer$ decision generates a NF_ACCEPT.

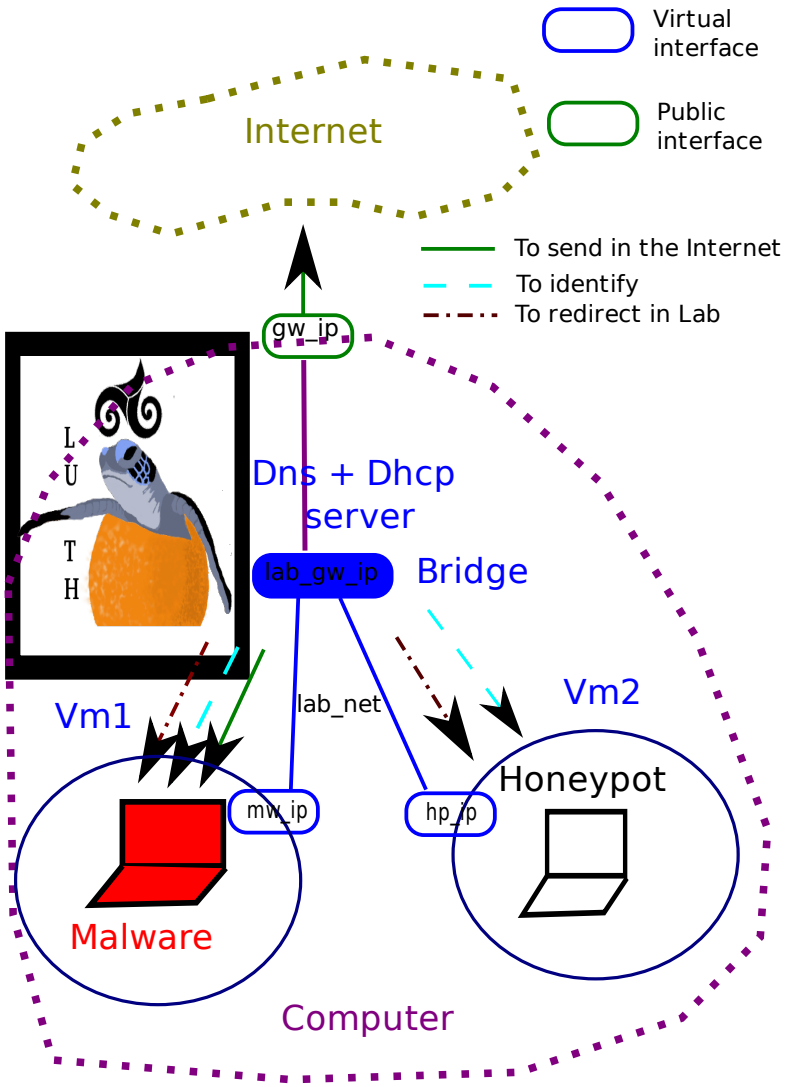


Figure 4.6: Platform to perform the dynamic monitoring of malware communications.

kqemu). We have not identified if this was due to a bug or to the fact that the malware detected the emulated environment. However, this constation was sufficient to choose KVM over Qemu.

The KVM version we use, 0.11.0, provides a very unstable snapshot feature. Therefore we implement the reinitializations of virtual machines by using two hard-drive images: an initial safe one and a current one. When rebooting the experiment, a copy of the initial safe image overrides the previous current image, which reinitializes the state of the virtual machine.

The Windows operating system is a XP-SP1. To implement the automatic execution of the malware, we install in this system a specially developed server, automatically launched at boot time. This server writes a binary sent by a client, in this case the software implementing the dynamic analysis, into a purposely created new file, and then executes it. The program implementing the experiment, which we call sandbox, drives among others Luth's reconfiguration. Once Luth is (re-) configured and launched, the sandbox sends the malware to analyze to the previously described server and waits for packets to inspect.

The honeypot's operating system is a Debian GNU/Linux 4.0. Both a Nepenthes instance and another specially developed server running on port 40000 are executed by the honeypot. This server accepts connections, and reads all the bytes sent over *TCP* until the connection is closed by the client. An iptables rule blocks all the request answers from the source port 40001 to implement the previously mentioned pit[13]. (NR3 in Listing 4.1)

The mac addresses of the virtual machine are configured with the option given during their process creation. The mapping between the IPv4 address, and mac addresses, both virtual and real, are written in */etc/ethers* of the host machine. The host computer has a Debian GNU/Linux 5.0 as operating system. It uses the dnsmasq [10] software as DHCP and DNS server. Both virtual machines are configured to use the DHCP protocol. The reader should notice that the DHCP messages are not monitored by Luth, which gives the opportunity to the malware to do whatever it wants there. Here again the development of DHCP MI is left for future work.

The software running the experiment, i.e. Luth, Luth reconfiguration, virtual machines executions and reboots, which we called sandbox, runs with root privileges. The virtual machines are executed using a specially created account with limited rights. The sandbox program stops the experiment by just killing the virtual machine processes, and reboots them using "fork" and "exec" system calls.

We deployed this environment in 5 computers in the laasnetexp platform in our laboratory, which presents the same characteristics depicted in Figure 2.11 (Section 2.3.2.1).

4.3.2 Results

This section describes the first two different results extracted from the data obtained with the latter platform. First, the manipulation of the recursively computed Luth rules enables us to classify malware. We analyzed 31 different malware downloaded from the 3 Nepenthes honeypot installed in 3 servers with 3 different IPv4 public addresses in our laboratory during one month between September and October 2009. During this period, these 3 servers downloaded binaries having 31 different MD5 sum hash[14].

We identify that among the 31 malware having a different MD5 sum hash, 27 belongs to 7 different botnets, the others being either:

- corrupted binaries,

[13]This aims at blocking the RST segments the TCP stack of the operating system will answer to any incoming segment to port 40001. This situation is analogous to the one described in Section 3.2.1.1

[14]We don't take into account the number of times a malware having the same MD5sum has been downloaded.

- quite nihilistic malware that just tries to propagate in the Internet like the blaster worm,
- member of inactive botnets.

Then, aiming to shutdown these botnets and identifying other hosts belonging to these ones, we show a way to translate part of Luth rules, into a set of a configuration of monolithic protocol layerings. Once this translation done, we provide scripts to configure less expressive but faster midpoint inspection devices like iptables [26] and snort [90].

Finally, we show how to use the dumps produced capturing the information of the dynamic analysis to evaluate the efficiency of already developed intrusion detection systems. To do so, we show how we mix a traffic dump present in the METROSEC database [192] available in our laboratory with different malware dumps.

4.3.2.1 Malware Clustering

The 31 malware samples are sequentially analyzed by the platform installed in four different hosts in our laboratory. Using the four different hosts, we simultaneously analyze 4 malware. The stopping of one analysis is done manually. Indeed, we periodically (about twice or three times a week) connect to each of the hosts to see how one analysis is performing[15]. Once we subjectively think one malware has given sufficient information, we stop the iteration, and start the analysis of the next malware. When stopping the analysis, among others the sandbox dumps the last computed rule in a file for further analysis. In this section we show how we use these rules to cluster malware.

The idea behind the clustering is to compare the inspection trees depicted in the dumped rule of each malware analysis. Indeed, the differences between all the rules reside in the dynamically added layerings. Therefore, if there is similarity between these new layerings, we consider the related malware to be similar.

To compare the inspection trees, we decide to first extract the dynamically computed part of the tree. Indeed, all the trees share the same initial rule (R and S nodes in Figure 4.7) and therefore, no useful information is present there. Then, we extract the paths of these dynamic trees (R and D nodes in Figure 4.7) by means of the paths function defined in [59]. These two operations are summarized in the Figure 4.7

We choose to cluster together the rules having a non empty intersection between the paths of the dynamically computed subtree. The rule representing a cluster is initially the one resulted from a malware execution. Then it becomes the union of its content with the ones of other rules having similar parts. The operation performed on two dynamic paths having a non empty intersection is depicted in the Figure 4.8. The Figure 4.9 depicts the operation on two dynamic paths having an empty intersection.

The algorithm can be summarized as depicted in Listing 4.9.

Listing 4.9: Clustering Algorithm

```
let rec cluster_one malware malware_path remaining_cluster clusters =
  if is_empty remaining_cluster then
    append_cluster clusters (create_cluster malware)
  else
    let choosed_cluster = choose remaining_cluster in
    let path_cluster = get_cluster_paths choosed_cluster in
    let inters = intersect malware_path path_cluster in
```

[15] This situation is indeed quite well illustrated by the comic in Figure 4.1.

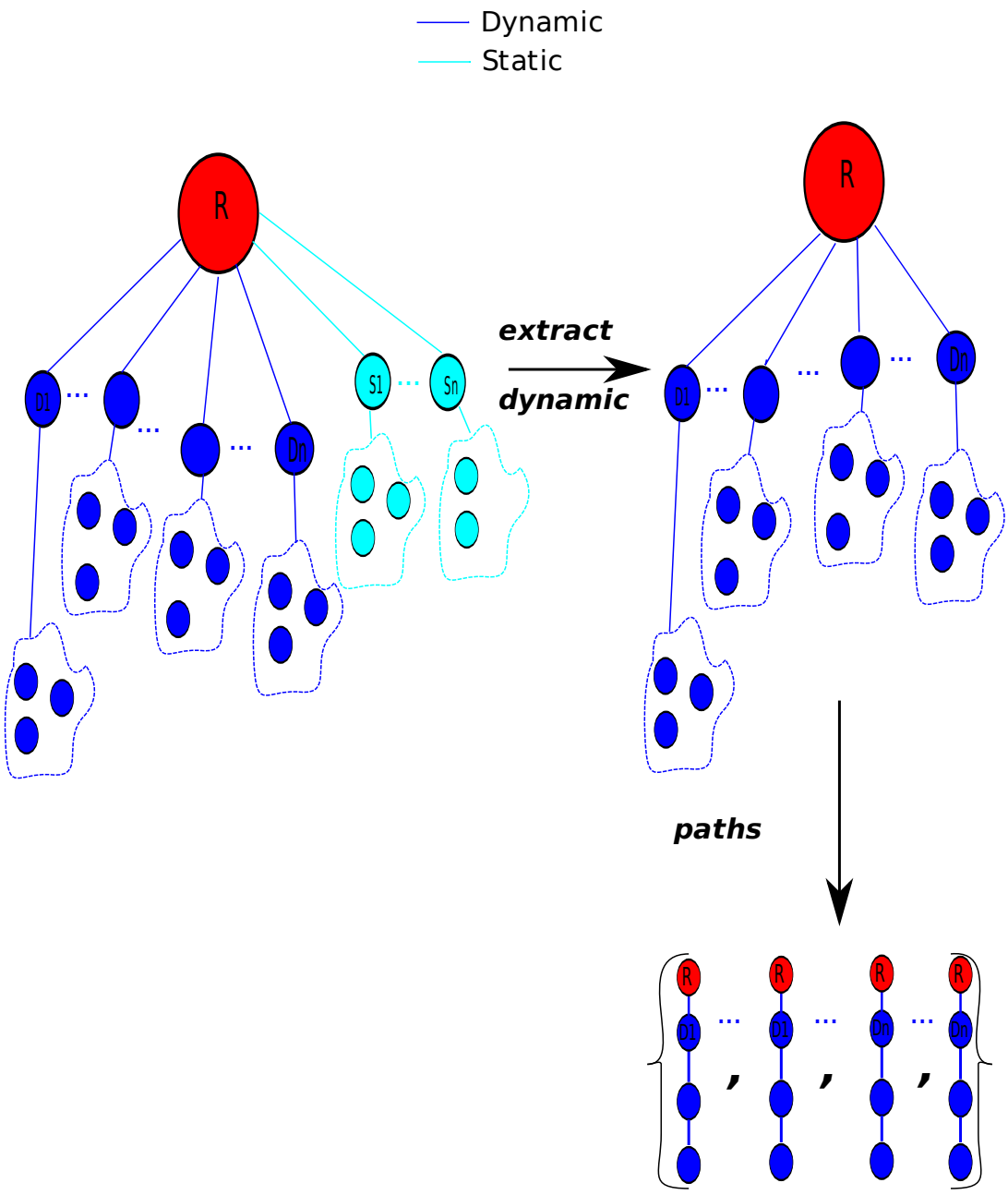


Figure 4.7: Processing dynamically computed layerings from the inspection tree.

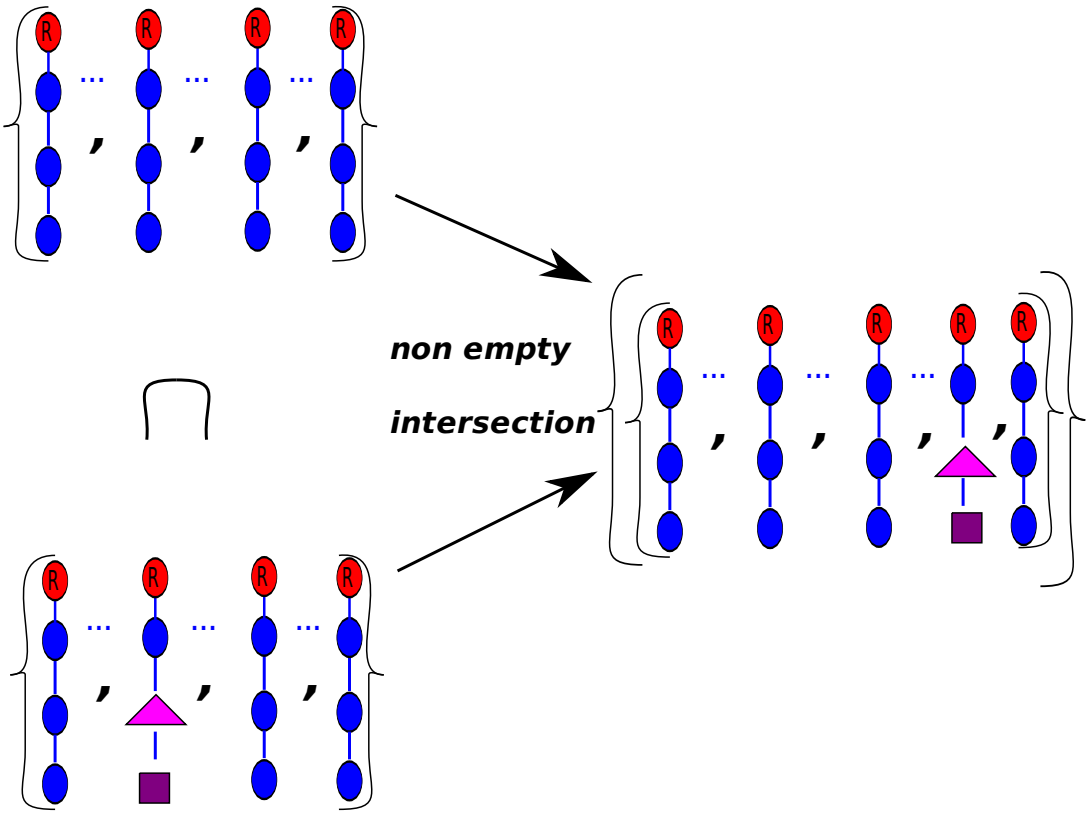


Figure 4.8: One iteration of the clustering process involving two dynamic paths having, at least, a path in common.

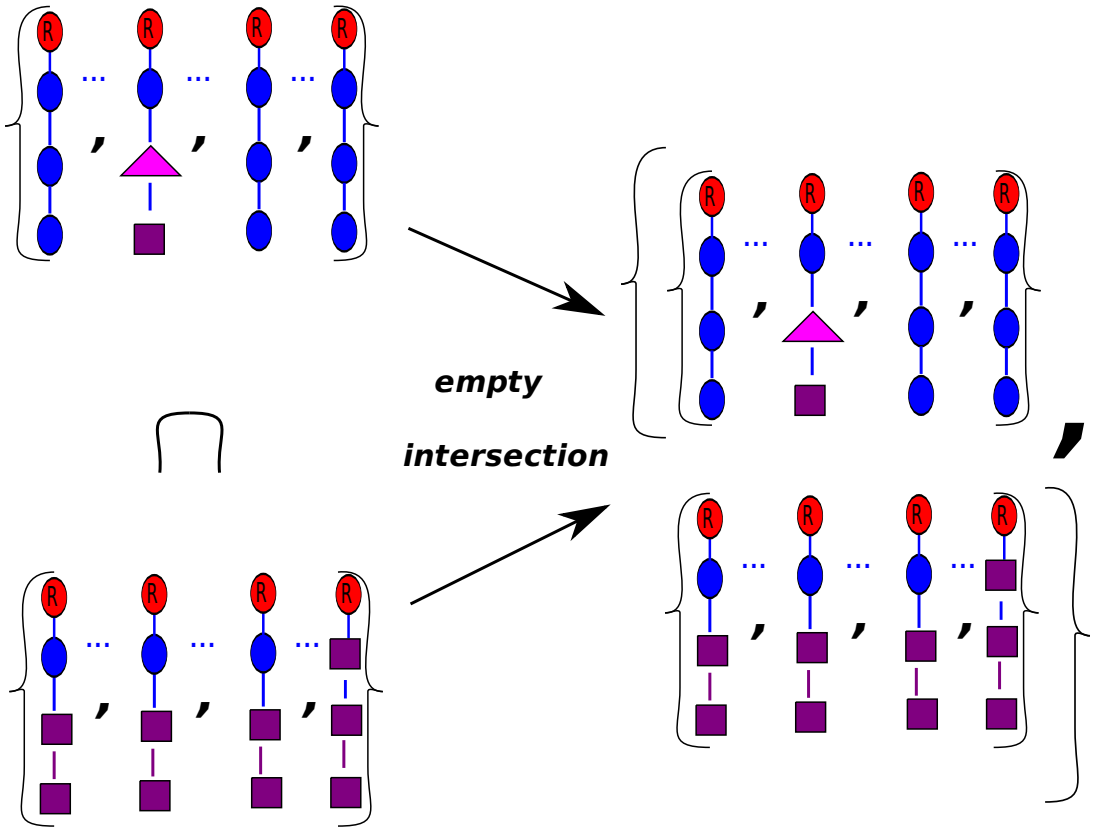


Figure 4.9: One iteration of the clustering process involving two disjoint dynamic paths.

```
      if is_empty inters then
        cluster_one malware malware_path
          (remove choosed_cluster remaining_cluster) clusters
      else
        let new_cluster_paths = join malware_path path_cluster in
        let clusters_without_choosed = remove choosed_cluster clusters in
        let new_choosed_cluster =
          append_malware_cluster clusters_without_choosed
           malware new_cluster_paths
        in
          append new_cluster_paths new_choosed_cluster;;


let rec cluster malware_set cl =
  if is_empty malware_set then cl
  else
    let one_malware = choose malware_set in
    let malware_path = get_malware_paths one_malware in
    let ncluster = cluster_one one_malware malware_path
      cl cl
    in
      cluster (remove one_malware malware_set) ncluster;;
```

This algorithm groups the malware, we represent by their MD5sum, into nine different clusters:

B0: 5ae700c1dffb00cef492844a4db6cd69, 879c6366be62dbc6492760a4d54d2c7b, 8a3f15b1be9b025bf8b3161939801a00, a894e6640a6e45d36fa7e7c1ba5d7a25, b7896907b97667eb1c9b307033e060fd, da02f21aadc011fb4d4528f7da3d3744, f97f1f583474f8cfb754aa449274047f, 1a2c0e6130850f8fd9b9b5309413cd00,

B1: 89041edc1c3d0b14b231575602ae1570,

B2: 8128405d8c32a75bab02a1f0d125d11c,

B3: 7dc73bfa4d78284155dd5101991eeb34,

B4: 1d419d615dbe5a238bbaa569b3829a23,

B5: 013a5ba10e3fc8a039b045530381d957,

B6: cc88f4f016cb52cceb6d9acfe271e233,

B7: 161651c69b69a2cd40784772f08cbd39, de2a8e3f8e782d0a4847446d6bb39601, aa03030d91d6931258190ba388685d85, 524bc0f75c12683f73ce0ceed70faab8, 524bc0f75c12683f73ce0ceed70faab8, 25930784ca2b8eed4d4c0461225c4a32,

B8: 14a09a48ad23fe0ea5a180bee8cb750a, fe234e9bd7bfcf2c4eb66303fa5abc4e, fd28c5e1c38caa35bf5e1987e6167f4c, f4a200f7818dfb166b9a3d238ac55a2d, e269d0462eb2b0b70d5e64dcd7c676cd, df51e3310ef609e908a6b487a28ac068, bb39f29fad85db12d9cf7195da0e1bfe, 833cda5b5bef5989deb6bf57c557ce30, 7b213ea370cc70aa4a72e6f16060aa17, 697f001bc330ab483d77a707cd40c8d9, 2fa0e36b36382b74e6e6a437ad664a80,

The B0 group represents the malware that has not any dynamically computed layering. In other words, these malware neither speak the *IRC*, nor the *HTTP* dialects the *IrcId* and *HttpId* MI would have recognized during the different iterations. As depicted by the analysis reported by the Virus Total website (the links can be found in Appendix 4), the 1a2c0e6130850f8fd9b9b5309413cd00 is recognized as the Sasser worm, the 5ae700c1dffb00cef492844a4db6cd69, as the Blaster worm, and 8a3f15b1be9b025bf8b3161939801a00 as the Dabber worm. These worms just scan and try to propagate on other systems targeting vulnerabilities on SMB services, or checking the presence for trojan horses at predefined ports, without waiting for orders. This explains why they do not speak among others, neither *IRC* nor *HTTP*, and therefore why they belong to the B0 group. The other binaries belonging to the B0 group: a894e6640a6e45d36fa7e7c1ba5d7a25, da02f21aadc011fb4d4528f7da3d3744, f97f1f583474f8cfb754aa449274047f, 879c6366be62dbc6492760a4d54d2c7b are incomplete versions of some malware binary, obtained by an interrupted download done by the nepenthes honeypot. These binaries are incomplete, do not execute correctly and are therefore harmless. Nevertheless, it is interesting to see that among the different antivirus detection engines some detect them as viruses, when others don't. We could see that for example when analyzing da02f21aadc011fb4d4528f7da3d3744, 14 antiviruses rise false positives among 41. These results validate the ones described in [88]. Finally, the b7896907b97667eb1c9b307033e060fd binary is surely part of a dismantled botnet using *IRC* Command & Control channel. Indeed, the asked *DNS* resolution to one of the probable servers fails and therefore the malware does nothing.

The parameters inside the dynamically created MI layerings involve *IPv*4 addresses, and *TCP* ports. as illustrated by *NMI* in Listing 4.4. Therefore, even two NMI layerings having different *IPv*4 addresses belonging to the same *DNS* label and the same destination *TCP* port are be considered as different. However, making a *DNS* label point to different servers at different time can be useful for furtivity and high availability reasons. Indeed, botnets are known to use different techniques grouped in the generic term: "Fast-flux" which use the DNS protocol to improve their network's furtivity and high availability [182]. That's the reason why we check manually, by using the *DNS* informations dumped by the Dns MI, the corresponding between the *IPv*4 addresses and *DNS* labels in the different clusters. We identify that 8128405d8c32a75bab02a1f0d125d11c (B3) and 013a5ba10e3fc8a039b045530381d957 (B5) refer to the same *DNS* label (DRD3H.com) on the same *TCP* port (6668) when speaking *IRC*. Similarly 1d419d615dbe5a238bbaa569b3829a23 (B7) and the binaries in (B9) talk *IRC* to the same *DNS* label (botz.noretards.com) in *TCP* port 65146. Therefore, if we consider the *DNS* label instead of *IPv*4 addresses in the NMI, the algorithm creates 7 groups instead of 9.

We have shown in this section how we use the rules computed by our dynamic analysis scheme to cluster malware. We group into nine different clusters the 31 malware to analyse. These clusters could be more precise by developing more adapted MI, like:

- vulnerability identifications MI to differentiate Sasser, Baggle and Dabber worms from non working malware, or
- using *DNS* label in *TPort* parameter of NMI instead of IPv4 addresses.

We do think the reverse engineering of a malware is a necessary first step to have a deep understanding of their behaviour. However, so that similar enough malware are not re-reverse engineered again, this clustering scheme could be used. To do so, once the new malware is reverse-engineered, corresponding ID MI should be developed in a Luth like framework. Then, by running the new malware sample in this platform, the similarity with already developed malware can quite easily be evaluated. In this example, by more or less developing 2 ID MI, (the IRC MI are indeed very similar), we have analysed all the

malware collected during a 1 month malware collection on three different servers, without reverse engineering any sample. The quite negligible effort to develop these MI[16] seems profitable to us. One could say this approach is useless because nowadays quite a lot of botnets use encyphered communications. We have two arguments to balance the latter sentence:

- Some studies provide ways to detect encyphered communications without having to uncypher them [75, 198]. Once this layering is identified, the following question rises: during the next iteration, is it safe to let the malware use this channel in the Internet, even if Luth cannot decypher it ? This can of behaviour could be configured by a layering like

  ```
  (..)/Tcp/TPort/StatisticallSSLId(
  new_rule_layering=(..)/Tcp/TPort(<previous>)/Void)
  ```

  This question makes us come back to the debate mentioned in the end of Section 4.1.3.3. Indeed, if we take the example of malware belonging to the B8 group, once connected to the IRC server having the address 72.10.172.2111 and TCP port 8080, the latter receives an order[17] that makes the malware connect twice to 67.215.1.206 address on TCP port 80 to download two binaries with the requests:

  - `GET /rr2.exe HTTP/1.0\r\nHost: idc.info\r\n,`
  - `GET /f4.exe HTTP/1.0\r\nHost: idc.info\r\n.`

  After the first of the two binaries is downloaded and executed, the server connects to the IRC server 67.43.236.67 on TCP port 1863. Without the dynamically computed Irc and Http layerings which authorizes the latter communications, these behaviours would not have been accepted by Luth. Without authorizing them, the quantity of information extracted by the dynamic analysis would have been reduced to the identification of the first IRC server on address 72.10.172.2111 and TCP port 8080. However, even if reduced, for clustering purposes, the identification of this server is sufficient to differentiate B8 from others groups. Finally, we do think the configuration of the ID rules is analogous to the calibration of the ordinate of the honeypots classification depicted in Figure 1.3. Indeed the configuration of the interaction can be adapted to some experiments while being insufficient for the goals aimed by other ones.

- As depicted in the work done by Chiang et.al in [98], the key used to encypher communications can be extracted from the malware binary. Moreover, they motivate the automation of this extraction in their perspectives. Therefore, once a new malware has been reverse engineered, a program to automate key extraction could be developed. Then, using this program, the key could be extracted and put as parameter of an MI that takes cyphered information, and outputs decyphered data to its children. A new and similar enough malware's network communications could therefore be analyzed without being re-reverse engineered.

To sum up, we do think that this platform helps automate the malware analysis the same way Unpackers, Debuggers, or other sandboxes do. Moreover, it is the first platform we are aware of to permit the automation of malware network communication analysis. Moreover, the information extracted by this experiment can help to automatically configure other midpoint inspection devices too.

[16] 581 lines of OCaml code.

[17] fKFeOdJTJX2ItzDq5pPlEkOW+StGxKOwCOjgJEzwGxXIF4lJwmrnrI/40Lk2U1jg8
/dyyq27LdDsuXEmX0OWIVnKQskYWhFgByfgP1dQBRfIN5tE9+RjpLv50ZsB4Mjg/
1nT+JyDJFB4qWEtUcOqJEPmzo1WI1AePlgz68A8h.

4.3.2.2 Signature Generation

In this section we describe how we use the dynamically computed layerings to configure two midpoint inspection devices: Netfilter and Snort. To do so, the problem consists in translating a Luth rule, into the language used to configure an already existing midpoint inspection device. As depicted in Section 2.1.2, such languages are based on the configuration of a set of *monolitic protocol layerings*. We first describe how compute these translations by formalizing them as a Tree pattern matching problems. Then, we evaluate the rules generated to configure the Snort software by performing an offline analysis. This analysis is performed on a trafic dump containing the network traces of five different malware mixed into a dump containing all the Internet conversations performed by the computers in our laboratory during one hour. The results show that the alarms output by Snort correctly detect the computers infected by the five malware present in the network dump.

Two problems need to be solved when trying to translate a rule in Luth into a rule in the language to configure other midpoint inspection devices:

Problem 3 *To compute the translation, the following two questions need to be answered:*

1. *Is the translation possible ?*
2. *How to compute possible translation ?*

This section does not address these problems generally, but provides a partial solution based on pattern matching in Trees [134]. Let us first remind the definition of the pattern matching problems.

As depicted in [134], the problems needs first to define a finite ranked alphabet Σ. Σ contains some elements of rank:

- 0, that can be seen as variables, like a,
- $q > 0$, that can be seen as functions with q arguments, i.e $a(arg_1, ..., arg_q)$.

Once this alphabet is defined, the set S of *Sigma-terms* is introduced.

Definition 9 *Let S be the set Sigma-terms defined like this:*

- $\forall b \in \Sigma$ *of rank 0, b is a* Σ*-term.*
- *If a is a symbol of rank q in* Σ*, then* $a(t_1, ..., t_q)$ *is a* Σ*-term provided each of the* t_i *is.*
- *Nothing else is a* Σ*-term.*

These Σ-terms represent labeled ordered trees. For example $a(a(b,b),b)$ represents the tree depicted in Figure 4.10. Once the trees are defined, we need a language to express patterns. To do so, the ν symbol is added to the Σ alphabet. Then, the S_ν is defined as set of $\Sigma \cup \{\nu\}$-terms. The ν symbol is used to express the "unknown parts" of the pattern. For example the pattern $a(\nu, b)$ matches the tree $a(a(b,b),b)$, at the nodes:

- $a(a(b,b),b), \nu = a(b,b)$ and,
- $a(b,b), \nu = b$.

Once we have S and S_ν, the matching problem can be defined.

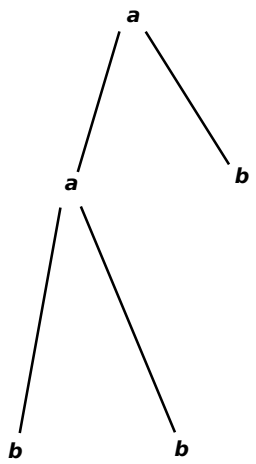


Figure 4.10: Representation of the a(a(b,b),b) Σ-term.

Definition 10 *A matching problem consists of a finite set of patterns $p_1, ..., p_k \in S_\nu$ and a subject tree $t \in S$. A solution to a matching problem is a list of all the pairs (n,i), where n is a node in t and p_i matches at n.*

Let us call *match_pattern* an algorithm solving this problem. The previous example can be summarized like that: $match\_pattern(\{a(\nu,b)\}, a(a(b,b),b)) = [(a(a(b,b),b),1);(a(b,b),1)]$.

We aim at providing a specific solution to the problem of the translation of a Luth rules by:

- First, translating a Luth rule into a Σ-term,
- Then, choosing the patterns of Luth rules that we want to translate into the configuration of a set of monolithic protocol layerings.

The case study we choose to illustrate this functionality is the creation of a Network Virus Detection system, a particular case of a Network Intrusion Detection System (NIDS). We do so by translating the previously mentioned clustered dynamically computed layering, into a file that configures the Snort NIDS. To show the genericity of the approach, we translate these rules into a list of iptables rules as an illustrative example.

Once the clustering process described in Section 4.3.2.1 is done, we have a set of clusters, identified by a set of dynamically computed layerings. We use in the following lines the same labels given to the clusters as in the previous section.

To obtain a Σ-term we first group into a tree, *DynTree*, the set of dynamically computed layerings by means of the join function depicted in Section 2.2.2.5. Then, we define that an MI present in *DynTree* is of rank q=m+n, m being the number of arguments of the MI, and n the number of its children. The

order at which the arguments of an MI are put is the same as the one they are listed in the manual. Both operations are depicted in Figure 4.11. The translation of the boolean expressions present, among others, in the arguments *cl_addr*,*sr_addr*,*cl_port* and *sr_port* of the *TPort* MI, requires a special attention. The configuration languages of Snort and Iptables do not support expressions like ((192.168.0.1 or 192.168.4.3) && (192.168.0.0/24)). Said otherwise, rules like

- iptables -A OUTPUT -s ((192.168.0.1 or 192.168.4.3) && (192.168.0.0/24)) -j DROP, or
- alert tcp ((192.168.0.1 or 192.168.4.3) && (192.168.0.0/24)) any -> any any.

are invalid. That's why we need to avoid such kind of situations during our translations. To be able to express by means of a pattern that we do not want to match boolean expressions, we translate the boolean expressions into Σ-terms too, following the definition depicted in Section 4.1.3.2. For example $(192.168.0.1\ or\ 192.168.4.3)\ \&\&\ (192.168.0.0/24)$ is translated into $And(Val(192.168.0.0/24), Or(Val(192.168.4.3), Val(192.168.0.1)))$.

Using this scheme, the tree[18] :

```
TPort(sr_port=Val Any, ds_port=Val Any,
      cl_addr=Val Any, sr_addr=Val Any,
      do_frag=yes)/
[ CSend(tm=None, do_frag=yes) |
  Http(tm=None, check_host=generic) ]
```

is translated into the Σ-term:

```
TPort(sr_port(Val(Any)), ds_port(Val(Any)),
      cl_addr(Val(Any)), sr_addr(Val(Any)), do_frag(yes),
      CSend(tm(None), do_frag(yes)),
      Http(tm(None), check_host(generic)))
```

Taking into account the syntax of the Snort and iptables commands, we choose to look for the following patterns in the computing tree:

P_1: `TPort(sr_port(Val(v)), ds_port(Val(v)),`
`cl_addr(Val(v)), sr_addr(Val(v)), do_frag(v),`
`Irc(v,v))`

P_2: `TPort(sr_port(Val(v)), ds_port(Val(v)),`
`cl_addr(Val(v)), sr_addr(Val(v)), do_frag(v),`
`Http(v,v))`

The reader should notice that the patterns Val(v) permits us to avoid matching the boolean expressions we mentioned earlier. The last thing to do, is to translate the list of (n,i) couples, output by *match_algorithm*, i.e $matched = match\_algorithm(\{P_1, P_2\}, To\Sigma\text{-}term(DynTree))$.

The answer to the first question of the Problem 3 is done that way: if *matched* is empty the translation is not possible. The answer to the second question, is done by translating the nodes matched by the patterns P_1 and P_2 contained in the list *matched*. To do so, we extract the destination address, $< daddr >$,

[18] It should be noticed that these trees are the abstract syntax trees computed by the interpreter. Indeed the user do not directly write $TPort(sr\_port = Val\ Any)$, but $TPort(sr\_port = *)$. The interpreter then translates $TPort(sr\_port = *)$ into $TPort(sr\_port = Val\ Any)$.

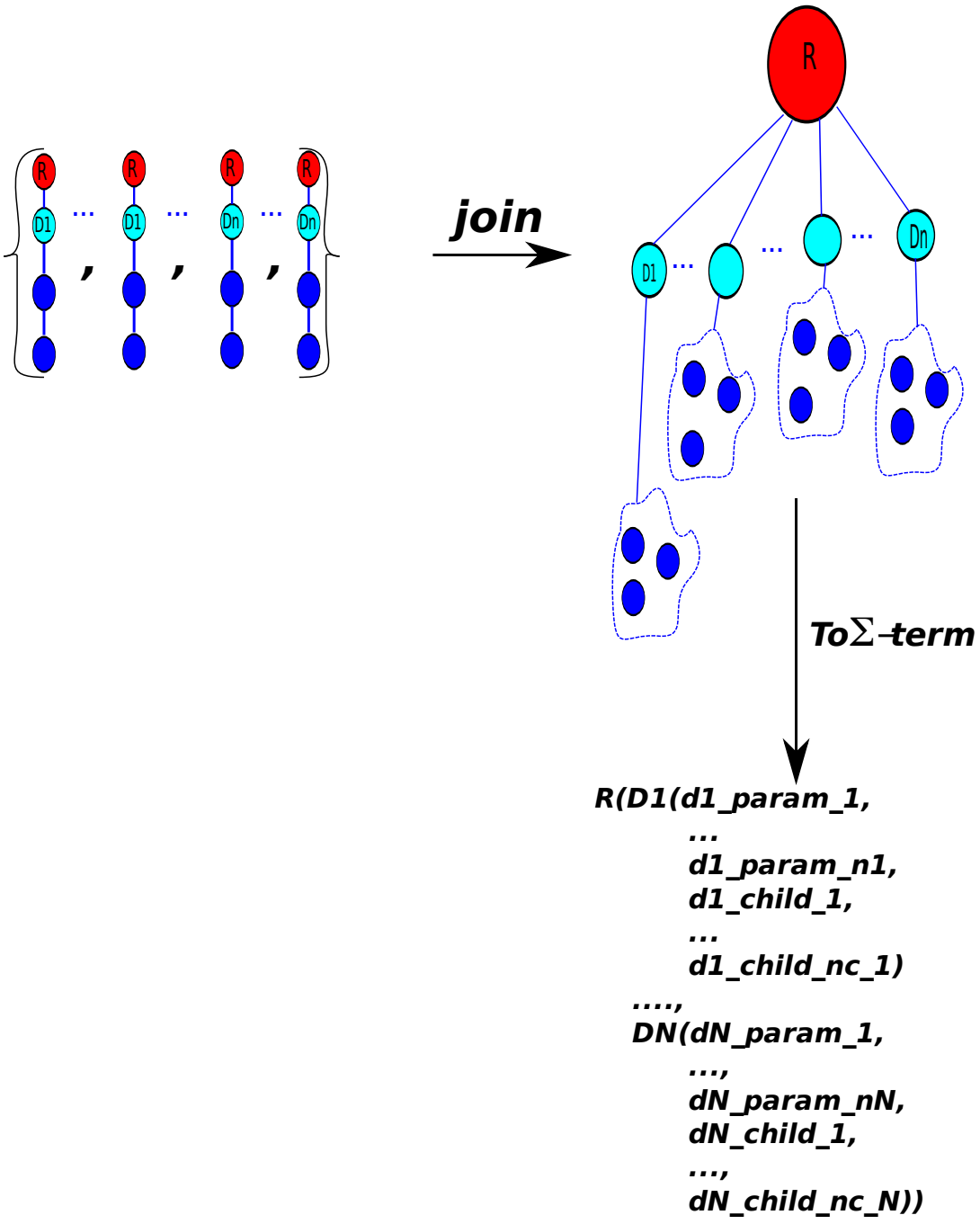


Figure 4.11: Computing the $\Sigma - term$ of the computed trees identifying clusters.

and the destination port, $< dport >$, of the *TPort* MI in the nodes present in the *matched* list. Then, we assign the string variable, "*C&C*" like "Command & Control" to the nodes matched by P_1, i.e of the form (n,1) in *matched*, and "*Dw*" like "Download", by the nodes matched by P_2, i.e of the form (n,2) in *matched*. Finally the variable "*Bx*" contains the labels assigned to the dynamically computed layering. The iptables and Snort rules are finally computed using the following format:

```
#iptables format
iptables -t filter -I IPS 1 -d <addr> -p tcp --dport <dport>\
-j LOG --log-prefix "Botnet <Bx>: <C&C|Dw>"
iptables -t filter -I IPS 1 -d <addr> -p tcp --dport <dport>\
-j DROP
```

and

```
#snort format
alert tcp any any -> <daddr> <dport>\
(msg: "Botnet <Bx>: <C&C|Dw>"; classtype: trojan-activity; sid:0; rev:1;)
```

Even if interesting, the tree pattern matching solution does not give a correct solution to the Problem 3. For example, let us go back to the problem of translating the $(192.168.0.1$ *or* $192.168.4.3)$ && $(192.168.0.0/24)$ expression that can be present in for example the *TPort* MI. We said that as the syntax of snort and Iptables do not give the possibility to express such a boolean expressions, we don't translate such a rule. However, when evaluating the latter expression, we can see that $((192.168.0.1$ *or* $192.168.4.3)$ && $(192.168.0.0/24)) \simeq 192.168.0.1$ and that therefore the expression could be translated into the Σ-term *Val* $(192.168.0.1)$ which can be translated into Snort and Iptables rules the algorithm based on pattern matching. The same way, the reader should notice that the rule:

```
TPort/(cl\_addr=192.168.0.1 or 192.168.4.3)/Irc
```

could be translated, into the following snort rule:

```
alert tcp 192.168.0.1 any -> any any\
(msg: "Botnet <Bx>: C&C"; classtype: trojan-activity; sid:0; rev:1;)
alert tcp 192.168.4.3 any -> any any\
(msg: "Botnet <Bx>: C&C"; classtype: trojan-activity; sid:0; rev:1;)
```

By this examples we show that solving the translation problem is more complicated than matching some patterns in Luth's inspection tree. However, even if the answer to the second item of the Problem 3 is partial, the computed translations provide useful results.

We evaluate the automatically generated Snort rule by asking snort to analyze a network traffic dump. We want this traffic dump to have both:

- traffic generated by some instance of the malware we analyzed, to evaluate if the rules correctly detects malware[19], and
- traffic generated by computers a-priori non infected by such malware, to evaluate if the rules do not identify a-priori non infected hosts as infected[20].

The traffic generated by some instance of the malware is available by means of the *L2N* MI used during the dynamic analysis. Indeed, during each iteration of the analysis, a new dump is created by

[19] In other words the measure the false negative-s.

[20] In other words to measure the false positive-s.

Trace	Packet capture length	Dump Size	First packet's time-stamp	Last packet's time-stamp
Laboratory	60B	481MB	2006-03-19 21:52:27	2006-03-19 22:52:25
$MB2$	All	309MB	2009-11-15 21:38:46	2009-11-17 8:42:19
$MB6$	All	2.9KB	2009-11-27 9:57:29	2009-11-27 9:58:54
$MB7$	All	574MB	2009-10-23 3:53:32	2009-10-24 4:01:09
$MB8_1$	All	40MB	2009-11-19 21:30:12	2009-11-19 21:32:9
$MB8_2$	All	51MB	2009-12-15 4:41:39	2009-12-15 6:31:58

Trace name	MD5sum of the malware that produced the trace
$MB2$	8128405d8c32a75bab02a1f0d125d11c
$MB6$	cc88f4f016cb52cceb6d9acfe271e233
$MB7$	aa03030d91d6931258190ba388685d85
$MB8_1$	833cda5b5bef5989deb6bf57c557ce30
$MB8_2$	fe234e9bd7bfcf2c4eb66303fa5abc4e

Figure 4.12: Different network traces used in the evaluation of the Snort rule.

this MI. We arbitrarily choose five different malware dumps to conduct our experiment. The traffic generated by computers a-priori non infected by such malware is available by means of the network captures done during the METROSEC project in our laboratory [192]. Among the different traces, some dumps represent the activity of all the computers in our laboratory. The characteristics of the traces we select are depicted in Figure 4.12.

The packet capture length shows the maximum size of the packets allowed to be captured in the dump. As depicted in Figure 4.12 the Laboratory dump contains at most 60 byte of an observed packet. However, the *L2N* MI chooses to dump the entire packet in the trace. That's why the $MB < i >$, that stands for Malware in B_i, dumps contain all the bytes of the packets.

To create a single realistic dump from all this dumps, we first need to modify the $MB < i >$ dumps. Indeed, given the configuration of our dynamic malware analysis, the packets from and to the destination of the analyzed malware have the same *mw_ip* address (Figure 4.6) in the $MB < i >$ dumps. If we merge the dumps without changing these addresses, we can not for example differentiate a host executing the 8128405d8c32a75bab02a1f0d125d11c malware from a host executing the aa03030d91d6931258190ba388685d85 malware. Therefore, we provide a way to change the packets having the *mw_ip* address as source or destination, into mw_ip_i, and compute the different related checksums again. Another problems is related to the time-stamps. As depicted in Figure 4.12 the time-stamps of all the dumps are separated in time. Therefore, to create a dump where malware are simultaneously executed, we choose to shift the time stamps of the $MB < i >$ so that the first packet of $MB < i >$ arrives at $shift_i$ seconds after the first packet of the Laboratory dump. Finally, the last problem deals with the packet capture length. We choose that the packets in the new dumps have at most the least capture length of all the dumps, which in this case is 60 byte. We create a utility that makes all these modifications to a traffic dump. That way we produce the dumps depicted in Figure 4.13.

Finally we merge the Laboratory, $MB1'$, $MB2'$, $MB6'$, $MB7'$, $MB8'_1$ and $MB8'_2$ dumps into a dump named "snort_test", containing 18707296 packets, by means of the "mergepcap" utility available in the

Trace name	mw_ip_i	$shift_i$ (s)
$MB2'$	200.0.0.1	60
$MB6'$	200.0.0.2	120
$MB7'$	200.0.0.3	180
$MB8'_1$	200.0.0.4	100
$MB8'_2$	200.0.0.5	80

Figure 4.13: Characteristics of the modifed network traces.

Botnet name	Number of alerts	IPv4 address
B2	3	200.0.0.1
B6	1	200.0.0.2
B7	908	?(too much alarms to browse manually)
B8	19	200.0.0.5 (11), 200.0.0.4 (8)

Figure 4.14: Characteristics of the different alerts.

package provided by the "Wireshark" software [48]. Then, we execute the snort software configured with the generated rule and obtain 931 alerts. The alerts are output in an log file, using the following format:

```
[**] [1:0:1] Botnet B8: C&C [**]
[Classification: A Network Trojan was detected] [Priority: 1]
03/19-21:53:50.155584 200.0.0.5:1032 -> 91.205.241.87:65146
TCP TTL:128 TOS:0x0 ID:126 IpLen:20 DgmLen:40 DF
***A**** Seq: 0x641A939A  Ack: 0xC513E9BF  Win: 0xFD20  TcpLen: 20
```

Here we can identify that the 200.0.0.5 host rose the alarm "Botnet B8: C&C.".

We browse the 931 alerts using text processing utilities like "grep" [21] or "Emacs" [11] to obtain the tabular depicted in Figure 4.14.

To verify the correction of all the alarms, we need to verify that all the alerts correctly identify the host executing a malware. The number of alerts of B2, B6 and B8 are low enough to be inspected manually. As depicted in Figure 4.15, the malware hosts are correctly identified. To verify that the 908 alerts of B7 identify the 200.0.0.3 host, we produce a new dump, *snort_test_prime*, that merges Laboratory, $MB1'$, $MB2'$, $MB6'$, $MB8'_1$ and $MB8'_2$ dumps. Said otherwise $snort\_test = snort\_test\_prime \cup MB7'$. We perform the same previous analysis on *snort_test_prime* with the results depicted in Figure 4.15. As depicted in the Figure 4.15, there are 0 alerts belonging to the B7 when analyzing *snort_test_prime*. Therefore, we conclude that all the B7 alerts in *snort_test* correctly identify the host executing the malware in MB7', i.e the one with the 200.0.0.3 IPv4 address.

We have not understood yet the relation between the number of the alerts output by Snort and the sequence of packets having the same session characteristics. We can say that an alert is not output each time a TCP segment have the parameters described by the configured alert. Indeed, let us take the example the rule that addresses B2 members by searching for sessions towards the TCP server on address 193.198.49.137 and port 6668. This rule outputs 3 alarms. However, there are 10858 TCP segments towards the "193.198.49.137" address on port 6668 as depicted by the output of the command

Botnet name	Number of alerts
B2	3
B6	1
B7	0
B8	19

Figure 4.15: Characteristics of the different alerts on snort_test_prime.

Botnet name	Number of alerts
B2	239
B6	3
B7	723
B8	11

Figure 4.16: Characteristics of the different alerts with the inverted configuration of snort.

```
>tcpdump -nr snort_test.pcap "src host  193.198.49.137 and src port 6668" |
wc -l
reading from file snort_test.pcap, link-type EN10MB (Ethernet)
10858
```

To try to have a better understanding of these alarms, we ask snort to analyze the dumps using an inverted configuration file. This inverted file is created using the pattern matching approach and outputting rules in the format:

```
alert tcp <daddr> <dport> -> any any\
(msg: "Botnet <Bx>: <C&C|Dw>"; classtype: trojan-activity; sid:0; rev:1;)
```

, instead of the previous one:

```
alert tcp any any -> <daddr> <dport>\
(msg: "Botnet <Bx>: <C&C|Dw>"; classtype: trojan-activity; sid:0; rev:1;)
```

We obtain the results depicted in Figure 4.16 with the new configuration file. Indeed, it is quite difficult for us to understand why for example B2 has 3 alerts in one direction, and 239 in the others. We assume that this trend is due to the notion of TCP statefulness implemented in Snort[21]. Therefore, the reader should understand that the number of alerts do not give us any useful information on the detection pattern. We just can say that all the alerts output by our configuration of Snort correctly identify infected hosts.

We introduced in this section the problem of the translation of Luth rule into other languages to configure other midpoint inspection devices based on the configuration of a set of **monolitich protocol layerings**. Even if it does not resolve the problem of the translation in general, we used an approach based on tree pattern matching that provides a partial solution. Using this pattern matching paradigm, we successfully translated the dynamically computed layerings identifying one cluster, into Snort and

[21]For the curious reader, we tried to disable this functionality with the directives "preprocessor stream4: noinspect", and "preprocessor stream4_reassemble: noalerts" depicted in the manual of Snort in vain: the results are the same.

Iptables configuration files. The evaluation of the Snort rule showed that, among others, we can generate network virus detection systems by translating dynamically computed layerings.

4.4 Conclusion

This last chapter addressed the original platform we design to perform malware's network communications dynamic analysis. The platform is based on a recursive approach that dynamically configures the next iteration of the experiment. To do so, we add two main different functionalities to Luth:

- The ability to perform basic Network Address Translation (Section 4.1.1)
- The *Id MI, that implement the New_rule decisions (Section 4.1.3.3).

The main originality of this platform is implemented by means of the *Id MI. Indeed by giving the possibility to configure as precisely as wanted the layering to use in the next iteration of the experiment, the user can configure the risks he is willing to take during the observation of the malware.

Then the first results obtained by the analysis of 31 different malware caught by 3 malware downloading honeypots during one month are depicted. The first case study addressed malware clustering (Section 4.3.2.1). By extracting the dynamically computed layerings during the different dynamic analysis of malware, we clustered the different malware into nine different groups we identified as botnets. Then we showed how the clustering could be more accurate by developing more adapted MI. Nevertheless, this first step already reduced the task of manual malware analysis. Then, we provided a technique to translate some Luth rules into other midpoint inspection device configuration language based on tree pattern matching (Section 4.3.2.2). We showed how this translation could be used to generate network virus detection systems.

We do think the prospectives of the use of this platform are quite vast. First, the platform should be stressed on more modern botware. We purposely chose to analyze quite simple botware in the first use of this sandbox for pragmatic reasons. Indeed, performing the reverse engineering of 0-day botware **alone** would have taken too much time in this Ph.D. This situation is different when considering the joint work of virus reverse engineering experts with a single Luth expert. Then, among the other applications of the information collected by the dynamic analysis, we do think of two use-cases:

Real time DDoS attack detection: Let us considerate the case where during the dynamically analysis process, one malware starts a syn-flood targeting one server. In this case, a MI similar to SPM can identify this attack and send this information to the administrators of one network. That way this administrator can more precisely monitor packets targeting the attacked server, to identify or block the other nodes perpetrating the distributed attack.

Network Intrusion Detection System evaluation: Thanks to the L2N MI, we automatically create the network traffic generated by the analyzed malware. This traffic can be used by the different NIDS to evaluate their performance using, automatically generated, up-to-date and **real** attacks traces containing an unprecedented quantity of information.

Conclusion

This Ph.D addresses the problems rose by the automation of distributed malicious activities in the Internet. We first showed that such activities are possible due to the fact that the problems related to the bugs in software are underestimated. Then, considering that the solutions to develop correct or fault tolerant system are not adopted by Internet end hosts and their software, this Ph.D proposes a new midpoint inspection device to monitor their behaviour. We first present the contribution done in this Ph.D, and then enumerate some prospectives.

Contributions

The design of this midpoint inspection device is focused on two main principles, its expressiveness and its correction. The expressiveness of this midpoint inspection device was obtained by the definition of a new inspection algorithm. This algorithm is based on an inspection tree configured by means of a new configuration language. This configuration language is based on a set of midpoint inspectors, MI, that can be composed and parallelized. The first version of the inspection algorithm only addresses static inspection trees. The analysis of the inspection algorithm permitted us to create the tools to:

- Verify the validity of a given inspection tree,
- Guessing the missing parts of an incomplete tree,
- Compute an optimized version of an inspection tree.

The second version of the algorithm permits to dynamically modify the initial static tree in function of the parameters of some of its nodes. This second version has not been analyzed yet. Indeed, the problematics involved in this situation have only been solved by the works published by Carette et.al among others [89] while we were writing this Ph.D. We then implemented a prototype, Luth, used to evaluate the presented ideas. Luth has been implemented in OCaml and is composed among others of:

- the interpreter of the language, and
- a first a set of MI, that can quite easily be augmented.

To benchmark the performances of Luth, we first evaluated it under two scenarios involving DNS and FTP protocol filtering. Moreover, both scenarios involved 20 self encapsulated GRE tunnels to give an overview of Luth's language expressiveness. We saw that even if Luth slowed down DNS and FTP communications, the filtering of communications was correct. Moreover, a policy more adapted to the DNS case study enabled us to significantly reduce the slow-down induced by Luth.

Then, we have tested the functionalities of Luth in two case studies. The first case study addressed the monitoring of malware downloading honeypots. First, to demonstrate the need for monitoring such endhost software, we design two intrusions to show how two malware downloading honeypots can be hijacked as x86 and SQL code injection attacks proxies. Then, by using the expressivity of Luth we provide a way to implement quite an original policy. Indeed, usually the monitoring of network communications aims at only accepting authorized and non malicious packets, or at least dropping all the malicious packets. However in this case, so that the honeypots can download malware some malicious packets must be accepted. However, other packets, like the generated download requests, can contain intrusion attacks as demonstrated by our x86 and SQL code injections. Therefore, these kind of malicious packets must be dropped. We implement this compromise by implementing two different Tcp MI. These

MI were computed using a slight modification of a formal method to synthesize a midpoint inspectors in function of endhosts model. We performed an offline analysis using a specific rule that shows that Luth succeeds in implementing this original policy.

The last study addressed malware's network communications dynamic analysis. This analysis recursively and dynamically disovers layerings by redirecting malware communications. A sandbox, which makes quite an extensive use of Luth analysis, was implemented to perform this analysis. This recursive analysis first tries to identify already non identifed communications by redirecting them to a honeypot in the platform. Once identified, by means of the configuration language of Luth a user can configure the rule to monitor the identified communications during the next iteration of the experimentation. We stressed the platform on the 31 malware having different MD5sums collected during a 1 month period on 3 malware downloading honeypots in our laboratory. By analyzing the dynamically computed layerings we first provided an algorithm to cluster malware. This algorithm grouped the 31 malware into 9 different groups. These groups correctly separated different active malware belonging to different botnets. Finally, we provided a way to translate these dynamically computed Luth rules into Snort and Iptables configurations by means of tree pattern matching. We used the generated configuration file of the Snort IDS to create a network virus detection system. This system was evaluated by inserting arbitrarily chosen five network dumps from our dynamic analysis into a dump representing normal traffic. During this experiment the created network virus detection system correctly detected all the infected hosts, without producing false positives.

Prospectives

The earliest work to do concerns the validation of the algorithm behind our midpoint inspection device. First, we should validate the proofs presented in this Ph.D by means of a proof-assistant like Coq. Then, we should analyse the second version of the inspect algorithm that permits to dynamically modify the initial inspection tree. In a third step, we should address the parallelization of this inspection algorithm, by proposing an algorithm and analyzing it. Then, we could implement another prototype and demonstrate the correction of the overall implementation by means of a proof-assistant too. Then we could try to identify how to optimize this prototype, and prove the modifications too.

Concerning the monitoring of malware downloading honeypots new MIs could be developed, in particular to monitor the emulated vulnerabilities. Then, we could see how to compute TCP and MTCP midpoint automatas with more than one retransmission.

Finally, the prospective of the platform to perform malware's network communications dynamic analysis are, as said earlier, quite vast. A team of malware reverse engineering experts together with a Luth expert could try evaluate the output obtained by the clustering and signature generation algorithms with the dynamic analysis of more evolved botware. The real time DDoS attack detection paradigm could be evaluated providing an analyzed malware receives such orders. Finally, the use of automatically generated malware traces could be used to help to automate the tuning and the evaluation of network malicious activities detection algorithms.

Bibliography

[1] Anubis. http://anubis.iseclab.org.

[2] Armadillo. http://www.siliconrealms.com/armadillo.html.

[3] Beware: Hackers at play. Newsweek magazine. September 1983.

[4] Botsniffer: Detecting botnet command and control channels in network traffic. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.8092.

[5] C. http://www.acm.uiuc.edu/webmonkeys/book/c_guide/.

[6] C++. http://www.cplusplus.com/.

[7] Caml light. http://caml.inria.fr/caml-light/release.fr.html.

[8] Camlidl. http://caml.inria.fr/pub/old_caml_site/camlidl/.

[9] Cwsandbox. http://www.cwsandbox.org.

[10] Dnsmasq. http://thekelleys.org.uk/dnsmasq/doc.html.

[11] Emacs. http://www.gnu.org/software/emacs/.

[12] F(sharp) at microsoft research. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/.

[13] Haskell. http://www.haskell.org.

[14] Ieee 802.1d ethernet bridging. http://sourceforge.net/projects/bridge/files/.

[15] Intel ia-32 architectures software developper's manual. http://www.intel.com/products/processor/manuals/index.htm.

[16] Java. http://java.sun.com/.

[17] Javascript. https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference.

[18] Kvm. http://www.linux-kvm.org/page/Main_Page.

[19] Linux netfilter hacking howto: Netfilter architecture. http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO-3%.html#ss3.1.

[20] Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology 2, Number 1 / August, 2006*, 35–50. Computer Science.

[21] Manual of grep. http://unixhelp.ed.ac.uk/CGI/man-cgi?grep.

[22] Metasploit project. http://www.metasploit.com.

[23] Mlpcap. http://www.drugphish.ch/~jonny/mlpcap.html.

[24] Mysql. http://www.mysql.com/.

[25] Nat behavioral requirements for tcp. http://www.ietf.org/rfc/rfc5382.txt.

[26] Netfilter, firewalling, nat and packet mangling for linux. http://www.netfilter.org/.

[27] Norman sandbox analyzer pro. http://www.normann.com.

[28] The objective caml system release. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[29] Pf: The openbsd packet filter. http://www.openbsd.org/faq/pf/.

[30] Php. http://php.net/index.php.

[31] Python. http://www.python.org/.

[32] Qemu open source processor emulator. http://www.qemu.org.

[33] Rfc 3330 (rfc3330): Special-use ipv4 addresses. http://www.faqs.org/rfcs/rfc3330.html.

[34] Rfc 3439. some internet architectural guidelines and philosophy. http://tools.ietf.org/html/rfc3439.

[35] Rfc 793: Transmission control protocol (tcp). http://tools.ietf.org/html/rfc793.

[36] Rfc 959: File transfer protocol. http://tools.ietf.org/html/rfc959.

[37] Rfc1459 - internet relay chat protocol. http://www.faqs.org/rfcs/rfc1459.html.

[38] Rfc1519: Classless inter-domain routing (cidr): an address ass. http://www.faqs.org/rfcs/rfc1519.html.

[39] Rfc2663: Ip network address translator (nat) terminology and c. http://www.faqs.org/rfcs/rfc2663.html.

[40] Rfc2784: Generic routing encapsulation (gre). http://www.faqs.org/rfcs/rfc2784.html.

[41] Ruby. http://www.ruby-lang.org/en/.

[42] Scapy. http://www.secdev.org/projects/scapy/.

[43] Snort inline. http://snort-inline.sourceforge.net/.

[44] Standard ml. http://www.smlnj.org/sml.html.

[45] Tcpdump. http://www.tcpdump.org/.

[46] Universal tap driver. http://vtun.sourceforge.net/tun/faq.html.

[47] Vmprotect. http://www.vmprotect.ru.

[48] Wireshark. http://www.wireshark.org/.

[49] Xen virtual machine monitor. http://www.cl.cam.ac.uk/research/srg/netos/xen/.

[50] Russia accused of unleashing cyberwar to disable estonia., May 2007.

[51] Global 'cyber spy' network revealed, March 2009. Al Jazeera.http://english.aljazeera.net/news/americas/2009/03/200933033044%96652.html.

[52] 2008, E. P. N. Y. T. A. Four convicted in sweden in internet piracy case. http://www.nytimes.com/2009/04/18/business/global/18pirate.html.

[53] ABAD, C., AND BONILLA, R. An analysis on the schemes for detecting and preventing arp cache poisoning attacks. In *Distributed Computing Systems Workshops, 2007. ICDCSW '07. 27th International Conference on* (June 2007), pp. 60–60.

[54] ABU RAJAB, M., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), ACM, pp. 41–52.

[55] AL-SHAER, E., EL-ATAWY, A., AND SAMAK, T. Automated pseudo-live testing of firewall configuration enforcement. *IEEE J.Sel. A. Commun. 27*, 3 (2009), 302–314.

[56] ALATA, E., ALBERDI, I., NICOMETTE, V., OWEZARSKI, P., AND KAÃĆNICHE, M. Internet attacks monitoring with dynamic connection redirection mechanisms. *Journal in Computer Virology 4*, 2 (2008), 127–136.

[57] ALATA, E., NICOMETTE, V., KAANICHE, M., DACIER, M., AND HERRB, M. Lessons learned from the deployment of a high-interaction honeypot. In *EDCC '06: Proceedings of the Sixth European Dependable Computing Conference* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 39–46.

[58] ALBERDI, I. Details of the userspace nat implementation. Tech. rep., LAAS-CNRS, 2010.

[59] ALBERDI, I. Formalization of the midpoint inspection device. Tech. rep., LAAS-CNRS, 2010.

[60] ALBERDI, I. Manual of luth. Tech. rep., LAAS-CNRS, 2010.

[61] ANONYMOUS. Once upon a free(). Phrack Magazine 57(9).

[62] APPEL, A. W. Book review: Garbage collection: Algorithms for automatic dynamic memory management by richard jones and rafael lins, john wiley & sons, 1996. *J. Funct. Program. 7*, 2 (1997), 227–229.

[63] BACKUS, J. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. 1977.

[64] BAECHER, P., KOETTER, M., HOLZ, T., DORNSEIF, M., AND FREILING, F. C. The nepenthes platform: An efficient approach to collect malware. In *RAID* (2006), pp. 165–184.

[65] BAILEY, M., OBERHEIDE, J., ANDERSEN, J., MAO, Z. M., JAHANIAN, F., AND NAZARIO, J. Automated classification and analysis of internet malware. *Lecture Notes in Computer Science 4637/2007* (august 2007), 178–197. Recent Advances in Intrusion Detection.

[66] BAILEY, M. D. *A Scalable Hybrid Network Monitoring Architecture for Measuring, Characterizing, and Tracking Internet Threat Dynamics*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 2006.

[67] BALAT, V. Ocsigen: typing web interaction with objective caml. In *ML '06: Proceedings of the 2006 workshop on ML* (New York, NY, USA, 2006), ACM, pp. 84–94.

[68] BARFORD, P., AND BLODGETT, M. Toward botnet mesocosms. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets* (Berkeley, CA, USA, 2007), USENIX Association, pp. 6–6.

[69] BART. Fsg: [f]ast [s]mall [g]ood exe packer. `http://www.xtreeme.prv.pl`.

[70] BARTAL, Y., MAYER, A., NISSIM, K., AND WOOL, A. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst. 22*, 4 (2004), 381–420.

[71] BASHE, C. Constructing the ibm ascc (harvard mark i). 65–75.

[72] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *NDSS 2009, 16th Annual Network and Distributed System Security Symposium, February 8-11, 2009, San Diego, USA* (02 2009).

[73] BEAUCAMPS, P. Advanced polymorphic techniques. In *PROCEEDINGS OF WORLD ACADEMY OF SCIENCE, ENGINEERING AND TECHNOLOGY* (2007), vol. 25.

[74] BEGEL, A., MCCANNE, S., AND GRAHAM, S. L. Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1999), ACM, pp. 123–134.

[75] BERNAILLE, L., AND TEIXEIRA, R. Early recognition of encrypted applications. vol. 4427/2007, Springer Berlin / Heidelberg, pp. 165–175.

[76] BHATTI, N. T., HILTUNEN, M. A., SCHLICHTING, R. D., AND CHIU, W. Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst. 16*, 4 (1998), 321–366.

[77] BHATTI, N. T., AND SCHLICHTING, R. D. Configurable communication protocols for mobile computing. In *ISADS '99: Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems* (Washington, DC, USA, 1999), IEEE Computer Society, p. 220.

[78] BIDDER-SENN, D., BASIN, D., AND CARONNI, G. Midpoints versus endpoints: From protocols to firewalls. In *ACNS '07: Proceedings of the 5th international conference on Applied Cryptography and Network Security* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 46–64.

[79] BINKLEY, J. R., AND SINGH, S. An algorithm for anomaly-based botnet detection. In *SRUTI'06: Proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet* (Berkeley, CA, USA, 2006), USENIX Association, pp. 7–7.

[80] BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones* (Oct. 2002), T. Mogensen, D. Schmidt, and I. Sudborough, Eds., LNCS 2566, Springer-Verlag, pp. 85–108.

[81] BLEXIM. Basic integer overflows, 2002. Phrack Magazine 60(10).

[82] BORELLO, J.-M., AND MÃL', L. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology 4*, 3 (2008), 211–220.

[83] BORIS V., C., ANDREW V., G., AND RADZIK, T. Shortest paths algorithms: Theory and experimental evaluation. 129–174.

[84] BORISOV, N., BRUMLEY, D., WANG, H., DUNAGAN, J., JOSHI, P., AND GUO, C. A generic application-level protocol analyzer and its language. In *14th Annual Network & Distributed System Security Symposium* (2007).

[85] BOS, H., DE BRUIJN, W., CRISTEA, M., NGUYEN, T., AND PORTOKALIDIS, G. Ffpf: fairly fast packet filters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 24–24.

[86] BROMLEY, A. G. Charles babbage's analytical engine, 1838. *IEEE Ann. Hist. Comput. 20*, 4 (1998), 29–45.

[87] CABRERA, J., LEWIS, L., QIN, X., LEE, W., PRASANTH, R., RAVICHANDRAN, B., AND MEHRA, R. Proactive detection of distributed denial of service attacks using mib traffic variables-a feasibility study. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on* (2001), pp. 609–622.

[88] CANTO, J., DACIER, M., KIRDA, E., AND LEITA, C. Large scale malware collection : lessons learned. In *IEEE SRDS 2008, 27th International Symposium on Reliable Distributed Systems. October 6-8, 2008, Napoli, Italy* (10 2008).

[89] CARETTE, J., KISELYOV, O., AND SHAN, C.-C. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program. 19*, 5 (2009), 509–543.

[90] CASWELL, B., FOSTER, J. C., RUSSELL, R., BEALE, J., AND POSLUNS, J. *Snort 2.0 Intrusion Detection*. Syngress Publishing, 2003.

[91] CC, C. Denial-of-service vulnerabilities in tcp/ip stacks. `http://www.cert.org/advisories/CA-2000-21.html`.

[92] CC, C. Dos attacks using nameservers. `http://www.cert.org/incident_notes/IN-2000-04.html`.

[93] CC, C. Smurf ip denial-of-service attacks. `http://www.cert.org/advisories/CA-1998-01.html`.

[94] CC, C. Tcp syn flooding and ip spoofing attacks. `http://www.cert.org/advisories/CA-1996-21.html`.

[95] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM Comput. Surv. 41*, 3 (2009), 1–58.

[96] CHANG, A., AND MERGEN, M. 801 storage: architecture and programming. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles* (New York, NY, USA, 1987), ACM, pp. 109–110.

[97] CHEN, X., ANDERSEN, J., MAO, Z., BAILEY, M., AND NAZARIO, J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on* (June 2008), pp. 177–186.

[98] CHIANG, K., AND LLOYD, L. A case study of the rustock rootkit and spam bot. In *HotBots'07* (Berkeley, CA, USA, 2007), USENIX Association, pp. 10–10.

[99] CHOI, H., LEE, H., LEE, H., AND KIM, H. Botnet detection by monitoring group activities in dns traffic. In *CIT '07: Proceedings of the 7th IEEE International Conference on Computer and Information Technology* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 715–720.

[100] CHRISTODORESCU, M., AND JHA, S. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2003), USENIX Association, pp. 12–12.

[101] CHRISTODORESCU, M., AND JHA, S. Testing malware detectors. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 2004), ACM, pp. 34–44.

[102] CHURCHILL, S. Iran government: Deep packet inspection? `http://www.dailywireless.org/2009/06/22/iran-government-deep-packet-ins%pection/`.

[103] CISCO. Ips solutions. `http://www.cisco.com/en/US/netsol/ns785/networking_solutions_package.ht%ml`.

[104] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (1986), 244–263.

[105] CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. model checking. *ACM Transactions on Programming Languages and Systems 16*, 5 (Sept. 1994), 1–45.

[106] CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. Model checking and abstraction. *ACM Trans. Program. Lang. Syst. 16*, 5 (1994), 1512–1542.

[107] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-end containment of internet worm epidemics. *ACM Trans. Comput. Syst. 26*, 4 (2008), 1–68.

[108] COUSOT, P. Abstract interpretation based formal methods and future challenges. In *Informatics - 10 Years Back. 10 Years Ahead.* (London, UK, 2001), Springer-Verlag, pp. 138–156.

[109] CROVELLA, M. E., AND BESTAVROS, A. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Trans. Netw. 5*, 6 (1997), 835–846.

[110] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[111] DENNING, P. J. Fault tolerant operating systems. *ACM Comput. Surv. 8*, 4 (1976), 359–389.

[112] DESWARTE, Y., ABGHOUR, N., NICOMETTE, V., AND POWELL, D. An internet authorization scheme using smart-card-based security kernels. In *E-SMART '01: Proceedings of the International Conference on Research in Smart Cards* (London, UK, 2001), Springer-Verlag, pp. 71–82.

[113] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), ACM, pp. 51–62.

[114] DREGER, H., FELDMANN, A., MAI, M., PAXSON, V., AND SOMMER, R. Dynamic application-layer protocol analysis for network intrusion detection. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2006), USENIX Association.

[115] DWING. Winupack 0.39 final. `http://dwing.51.net`.

[116] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic spyware analysis. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–14.

[117] EISENBERG, T., GRIES, D., HARTMANIS, J., HOLCOMB, D., LYNN, M. S., AND SANTORO, T. The cornell commission: on morris and the worm. *Commun. ACM 32*, 6 (1989), 706–709.

[118] EXPOSITO, E., SENAC, P., AND DIAZ, M. Fptp: the xqos aware and fully programmable transport protocol. In *Networks, 2003. ICON2003. The 11th IEEE International Conference on* (Sept.-1 Oct. 2003), pp. 249–254.

[119] F., P., M., D., AND H., D. Honeypot, honeynet,honeytoken: Terminological issues. Tech. rep., Eurecom, 2003.

[120] FLANAGAN, C. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM, pp. 245–256.

[121] FRANCILLON, A., AND CASTELLUCCIA, C. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), ACM, pp. 15–26.

[122] GATES, C., AND TAYLOR, C. Challenging the anomaly detection paradigm: a provocative discussion. In *NSPW '06: Proceedings of the 2006 workshop on New security paradigms* (New York, NY, USA, 2007), ACM, pp. 21–29.

[123] GERA, AND RIQ. Advances in format string exploiting, 2001. Phrack Magazine 59(7).

[124] GOEBEL, J., AND HOLZ, T. Rishi: identify bot contaminated hosts by irc nickname evaluation. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets* (Berkeley, CA, USA, 2007), USENIX Association, pp. 8–8.

[125] GOUDA, M., AND LIU, A. A model of stateful firewalls and its properties. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on* (June-1 July 2005), pp. 128–137.

[126] GOUDA, M. G., AND LIU, A. X. Structured firewall design. *Comput. Netw. 51*, 4 (2007), 1106–1120.

[127] GRAY, J. Dependability in the internet era. Stanford, 2006. `http://research.microsoft.com/en-us/um/people/gray/talks/internetdepend%abilitystanford_2006.ppt`.

[128] GRIZZARD, J. B., SHARMA, V., NUNNERY, C., KANG, B. B., AND DAGON, D. Peer-to-peer botnets: overview and case study. In *HotBots'07* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–1.

[129] GU, G., PERDISCI, R., ZHANG, J., AND LEE, W. Botminer: clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *SS'08: Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 139–154.

[130] GUO, F., FERRIE, P., AND CHIUEH, T.-C. A study of the packer problem and its solutions. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 98–115.

[131] HATHAWAY, M. Securing our digital future. `http://www.whitehouse.gov/cyberreview/`.

[132] HOARE, C. Viewpoint retrospective: an axiomatic basis for computer programming. *Commun. ACM 52*, 10 (2009), 30–32.

[133] HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (October 1969), 576–580 and 583.

[134] HOFFMANN, C. M., AND O'DONNELL, M. J. Pattern matching in trees. *J. ACM 29*, 1 (1982), 68–95.

[135] HOLZ, T., STEINER, M., DAHL, F., BIERSACK, E., AND FREILING, F. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–9.

[136] HSC. `http://www.hsc.fr/ressources/outils/dns2tcp/`.

[137] HU, X., CHIUEH, T.-C., AND SHIN, K. G. Large-scale malware indexing using function-call graphs. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), ACM, pp. 611–620.

[138] HUTCHINSON, N., AND PETERSON, L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (1991), 64–76.

[139] I., A., V., N., AND P., O. Beware of the use of malware downloading honeypot (case of study of nepenthes and php.hop). Tech. rep., LAAS-CNRS, 2007.

[140] ILGUN, K., KEMMERER, R. A., AND PORRAS, P. A. State transition analysis: A rule-based intrusion detection approach. *IEEE Trans. Softw. Eng. 21*, 3 (1995), 181–199.

[141] INSTITUTE, S. Top 20 internet security problems, threats and risks. section 5 anti-virus software. http://www.sans.org/top20/2007/#s5.

[142] INSTITUTE, S. The top cyber security risks. http://www.sans.org/top-cyber-security-risks/.

[143] JAISWAL, S., IANNACCONE, G., DIOT, C., KUROSE, J., AND TOWSLEY, D. Measurement and classification of out-of-sequence packets in a tier-1 ip backbone. *IEEE/ACM Trans. Netw. 15*, 1 (2007), 54–66.

[144] J.B, A. Computer security threat monitoring and surveillance. Tech. rep., Anderson Co. Fort Washington. PA, 1980.

[145] JIANG, X., XU, D., AND WANG, Y.-M. Collapsar: a vm-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *J. Parallel Distrib. Comput. 66*, 9 (2006), 1165–1180.

[146] JUNG, J., KRISHNAMURTHY, B., AND RABINOVICH, M. Flash crowds and denial of service attacks: characterization and implications for cdns and web sites. In *WWW '02: Proceedings of the 11th international conference on World Wide Web* (New York, NY, USA, 2002), ACM, pp. 293–304.

[147] JUNIPER. Safeguarding your network ips overview demo. http://www.juniper.net/products_and_services/intrusion_prevention_s%olutions.

[148] KAEMPF, M. Vudo malloc tricks, 2001. Phrack Magazine 57(8).

[149] KAMLUK, V. The botnet business, March 2009. http://www.viruslist.com/en/analysis?pubid=204792003.

[150] KANDULA, S., KATABI, D., JACOB, M., AND BERGER, A. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *In 2nd Symposium on Networked Systems Design and Implementation (NSDI* (2005).

[151] KEROMYTIS, A. D., AND SMITH, J. M. Requirements for scalable access control and security management architectures. *ACM Trans. Internet Technol. 7*, 2 (2007), 8.

[152] KING, S. T., AND CHEN, P. M. Backtracking intrusions. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 223–236.

[153] KIRDA, E., JOVANOVICH, N., KRUEGEL, C., AND VIGNA, G. Client-side cross-site scripting protection. *Computers and Security Journal, Elsevier, June 2009* (2009).

[154] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: formal verification of an os kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 207–220.

[155] KNUTH, D. Structured programming with go to statements. 257–321.

[156] KUMAR, S., AND SPAFFORD, E. H. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference* (1994), pp. 11–21.

[157] KUZMANOVIC, A., AND KNIGHTLY, E. W. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2003), ACM, pp. 75–86.

[158] LABIT, Y., AND MAZEL, J. Hidden: Hausdorff distance based intrusion detection approach dedicated to networks. In *ICIMP '08: Proceedings of the 2008 The Third International Conference on Internet Monitoring and Protection* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 11–16.

[159] LAKHINA, A., CROVELLA, M., AND DIOT, C. Diagnosing network-wide traffic anomalies. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2004), ACM, pp. 219–230.

[160] LANDI, W. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1*, 4 (1992), 323–337.

[161] LAPRIE, J. C., AVIZIENIS, A., AND KOPETZ, H., Eds. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[162] LEITA, C., MERMOUD, K., AND DACIER, M. Scriptgen: an automated script generation tool for honeyd. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 203–214.

[163] LEROY, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (2009), 107–115.

[164] LEVINE, J., MASON, T., AND BROWN, D. *Lex & yacc, 2nd edition*. O'Reilly, 1992.

[165] LIU, A. Formal verification of firewall policies. In *Communications, 2008. ICC '08. IEEE International Conference on* (May 2008), pp. 1494–1498.

[166] LIU, A. X., AND GOUDA, M. G. Diverse firewall design. *Parallel and Distributed Systems, IEEE Transactions on 19*, 9 (Sept. 2008), 1237–1251.

[167] LIVADAS, C., WALSH, R., LAPSLEY, D., AND STRAYER, W. Using machine learning technliques to identify botnet traffic. *Local Computer Networks, Annual IEEE Conference on 0* (2006), 967–974.

[168] LU, W., TAVALLAEE, M., AND GHORBANI, A. A. Automatic discovery of botnet communities on large-scale communication networks. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ACM, pp. 1–10.

[169] MADHAVAPEDDY, A., HO, A., DEEGAN, T., SCOTT, D., AND SOHAN, R. Melange: creating a "functional" internet. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 101–114.

[170] MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (Dec. 2007), pp. 431–441.

[171] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., AND BRUSCHI, D. Testing cpu emulators. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis* (New York, NY, USA, 2009), ACM, pp. 261–272.

[172] MCAFEE. Host intrustion prevention for server. `http://www.mcafee.com/us/enterprise/products/host\intrusion\prevention/host\intrusion\prevention\server`.

[173] MEYER, B. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[174] M.F, O., L, M., AND J.F., R. Upx: the ultimate packet for executables. `http://upx.sourceforge.net`.

[175] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (December 1978), 348–375.

[176] MIWA, S., MIYACHI, T., ETO, M., YOSHIZUMI, M., AND SHINODA, Y. Design and implementation of an isolated sandbox with mimetic internet used to analyze malwares. In *DETER: Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test 2007* (Berkeley, CA, USA, 2007), USENIX Association, pp. 6–6.

[177] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP '07. IEEE Symposium on* (May 2007), pp. 231–245.

[178] MUKHERJEE, B., HEBERLEIN, L., AND LEVITT, K. Network intrusion detection. *Network, IEEE 8*, 3 (May/Jun 1994), 26–41.

[179] MULLENDER, S. J., AND TANENBAUM, A. S. The Design of a Capability-Based Distributed Operating System. *The Computer Journal 29*, 4 (1986), 289–299.

[180] NACHENBERG, C. Computer virus-antivirus coevolution. *Commun. ACM 40*, 1 (1997), 46–51.

[181] NAMESTNIKOV, Y. The economics of botnets, March 2009. `http://www.viruslist.com/en/analysis?pubid=204792003`.

[182] NAZARIO, J., AND HOLZ, T. As the net churns: Fast-flux botnet observations. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on* (2008), pp. 24–31.

[183] NEUMANN, J. V. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[184] NIEVERGELT, J. On the time required for timing the halting problem rephrased. *IEEE Trans. Comput. 19*, 5 (1970), 458–459.

[185] NING, P., JAJODIA, S., AND WANG, X. S. Abstraction-based intrusion detection in distributed environments. *ACM Trans. Inf. Syst. Secur. 4*, 4 (2001), 407–452.

[186] NOH, S., LEE, C., CHOI, K., AND JUNG, G. Detecting distributed denial of service (ddos) attacks through inductive learning. In *Intelligent Data Engineering and Automated Learning* (aug 2003), vol. 2690/2003 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 286–295.

[187] NUSSBAUM L., R. O. Prototype de canal cachÃl' dans le dns. In *CFIP'08 : Colloque Francophone sur l'IngÃl'nierie des Protocoles, Les Arcs, France* (03 2008). `http://perso.ens-lyon.fr/lucas.nussbaum`.

[188] O.HOROVITZ. Big loop integer protection, 2002. Phrack Magazine 60(9).

[189] ONE, A. Smashing the stack for fun and profit, 1996. Phrack Magazine 49(14).

[190] ORMANDY, T. An empirical study into the security exposure to hosts of hostile virtualized environments. In *8th CanSec West Applied Technical Security Conference* (april 2007).

[191] OUDOT, L., ROPERT, F., AND RIDEN, J. Phphop. `http://www.rstack.org/PHPhop/`.

[192] OWEZARSKI, P. A database of anomalous traffic for assessing profile based ids. In *Second international workshop on Traffic Monitoring and Analysis (TMA'2010)* (2010), LNCS, pp. 59–72.

[193] OWEZARSKI, P., BERTHOU, P., LABIT, Y., AND GAUCHARD, D. Laasnetexp: a generic polymorphic platform for network emulation and experiments. In *TridentCom '08: Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities* (ICST, Brussels, Belgium, Belgium, 2008), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–9.

[194] PALME, J., AND OF COMPUTER SYSTEMS SCIENCE AT STOCKHOLM. JULY 2007., M. B. D. Anonimity on the internet. `http://people.dsv.su.se/~jpalme/society/anonymity.html`.

[195] PASSERINI, E., PALEARI, R., AND MARTIGNONI, L. How good are malware detectors at remediating infected systems? In *DIMVA '09: Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 21–37.

[196] PATCHA, A., AND PARK, J.-M. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw. 51*, 12 (2007), 3448–3470.

[197] PAXSON, V. Bro: a system for detecting network intruders in real-time. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium* (Berkeley, CA, USA, 1998), USENIX Association, pp. 3–3.

[198] PERÉNYI, M., AND MOLNÁR, S. Enhanced skype traffic identification. In *ValueTools '07: Proceedings of the 2nd international conference on Performance evaluation methodologies and tools* (ICST, Brussels, Belgium, Belgium, 2007), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–9.

[199] PIERCE, B. C. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[200] PINKAS, B., AND SANDER, T. Securing passwords against dictionary attacks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security* (New York, NY, USA, 2002), ACM, pp. 161–170.

[201] PINTO, A. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems* (Washington, DC, USA, 2001), IEEE Computer Society, p. 707.

[202] POLYCHRONAKIS, M., ANAGNOSTAKIS, K. G., AND MARKATOS, E. P. Real-world polymorphic attack detection using network-level emulation. In *CSIIRW '08: Proceedings of the 4th annual workshop on Cyber security and information intelligence research* (New York, NY, USA, 2008), ACM, pp. 1–3.

[203] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (1974), 412–421.

[204] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (New York, NY, USA, 2006), ACM, pp. 15–27.

[205] POULSEN, K. Fbi busts alleged ddos mafia. `http://www.securityfocus.com/news/9411`.

[206] PROVOS, N. A virtual honeypot framework. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 1–1.

[207] RAMESH, H., SANJIV, K., AND VIJAY, K. Faster enumeration of all spanning trees of a directed graph. 428–439.

[208] RANJAN, S., KARRER, R., AND KNIGHTLY, E. Wide area redirection of dynamic content by internet data centers. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies* (March 2004), vol. 2, pp. 816–826 vol.2.

[209] RATIU, I.-G., CARSTEA, C.-G., DAVID, N., AND BADAU, D. Antivirus software optimization. *WSEAS Trans. Info. Sci. and App. 5*, 7 (2008), 1095–1112.

[210] RAYMOND, E. S. How to become a hacker. `http://catb.org/esr/faqs/hacker-howto.html`.

[211] R.G., B. *Intrusion Detection*. Sams Publishing, 2000.

[212] RIECK, K., HOLZ, T., WILLEMS, C., DÜSSEL, P., AND LASKOV, P. Learning and classification of malware behavior. In *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 108–125.

[213] RIX. Writting ia32 alphanumeric shellcodes. Phrack Magazine 57(11).

[214] ROOIJ, G. V. Real stateful tcp packet filtering in ip-filter. In *Invited talk at the 10th USENIX Security Symposium*.

[215] ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LANGLOIS, S., LÃL'ONARD, P., AND NEUHAUSER, W. Overview of the chorus distributed operating systems. *Computing Systems 1* (1991), 39–69.

[216] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (1984), 277–288.

[217] SCHERRER, A., LARRIEU, N., BORGNAT, P., ABRY, P., AND OWEZARSKI, P. Non-gaussian and long memory statistical characterizations for internet traffic with anomalies. *IEEE Trans. Dependable Secur. Comput. 4*, 1 (2007), 56–70.

[218] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 552–561.

[219] SIGLER, L. E. L. E. *Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation*. Sources and studies in the history of mathematics and physical sciences. 2002.

[220] SOFTWARE, A. Aspack for windows. `http://www.aspack.com/aspack.html`.

[221] SPINELLIS, D. Reliable identification of bounded-length viruses is np-complete. *Information Theory, IEEE Transactions on 49*, 1 (Jan 2003), 280–284.

[222] STRAYER, W. T., LAPSLEY, D., WALSH, R., AND LIVADAS, C. Botnet detection based on network behavior. In *Botnet Detection: Countering the Largest Security Threat* (2007), W. Lee, C. Wang, and D. Dagon, Eds., Springer-Verlag.

[223] STUBBLEFIELD, A., IOANNIDIS, J., AND RUBIN, A. D. A key recovery attack on the 802.11b wired equivalent privacy protocol (wep). *ACM Trans. Inf. Syst. Secur. 7*, 2 (2004), 319–332.

[224] SU, Z., AND WASSERMANN, G. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM, pp. 372–382.

[225] SWADE, D., AND BABBAGE, C. *Difference Engine: Charles Babbage and the Quest to Build the First Computer*. Viking Penguin, 2001.

[226] SWAINE, J. Russia continues cyber war on georgia, August 2008. The telegraph. `http://www.telegraph.co.uk/news/worldnews/europe/georgia/2539157/Georgi%a-Russia-conducting-cyber-war.html`.

[227] TECHNOLOGY, O. Themida: Advanced windows software protection system. http://www.oreans.com/themida.php.

[228] TEUSCHER, C., AND SIPPER, M. Hypercomputation: hype or computation? *Commun. ACM 45*, 8 (2002), 23–24.

[229] THOMAS, C., AND JEAN-MARC, R. *Statistical Methods in Computer Security*. 2004.

[230] THOMAS, R., CHRISTOPHER, K., AND ENGIN, K. Detecting system emulators. In *Lecture Notes in Computer Science* (2007), vol. 4779/2007, Springer Berlin / Heidelberg, pp. 1–18.

[231] TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc. 2*, 42 (1936), 230–265.

[232] VILLAMARÍN-SALOMÓN, R., AND BRUSTOLONI, J. C. Bayesian bot detection based on dns traffic similarity. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing* (New York, NY, USA, 2009), ACM, pp. 2035–2041.

[233] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 148–162.

[234] VUTUKURU, M., BALAKRISHNAN, H., AND PAXSON, V. Efficient and robust tcp stream normalization. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 96–110.

[235] WANG, X., LI, Z., CHOI, J. Y., XU, J., REITER, M. K., AND KIL, C. Fast and black-box exploit detection and signature generation for commodity software. *ACM Trans. Inf. Syst. Secur. 12*, 2 (2008), 1–35.

[236] WILLIAMS, M. R. The origins, uses, and fate of the edvac. *IEEE Ann. Hist. Comput. 15*, 1 (1993), 22–38.

[237] WU, Z., XIE, M., AND WANG, H. Swift: a fast dynamic packet filter. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), USENIX Association, pp. 279–292.

[238] XIAO, B., CHEN, W., HE, Y., AND SHA, E. H. M. An active detecting method against syn flooding attack. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 709–715.

[239] XIE, Y., AND YU, S.-Z. Monitoring the application-layer ddos attacks for popular websites. *IEEE/ACM Trans. Netw. 17*, 1 (2009), 15–25.

[240] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 116–127.

[241] YOUNG, M., TEVANIAN, A., RASHID, R., GOLUB, D., AND EPPINGER, J. The duality of memory and communication in the implementation of a multiprocessor operating system. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles* (New York, NY, USA, 1987), ACM, pp. 63–76.

[242] YUAN, J. Monitoring the macroscopic effect of ddos flooding attacks. *IEEE Trans. Dependable Secur. Comput. 2*, 4 (2005), 324–335. Senior Member-Mills, Kevin.

[243] ZHANG, X., LI, C., AND ZHENG, W. Intrusion prevention system design. In *CIT '04: Proceedings of the The Fourth International Conference on Computer and Information Technology* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 386–390.

[244] ZIMMERMANN, H. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on 28*, 4 (Apr 1980), 425–432.

[245] ZOU, C. C., AND CUNNINGHAM, R. Honeypot-aware advanced botnet construction and maintenance. *Dependable Systems and Networks, International Conference on 0* (2006), 199–208.