ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2015-ENST-0059

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité Informatique et Réseaux

présentée et soutenue publiquement par

Mariano Graziano

le 28 Septembre 2015

Améliorations pour l'analyse mémoire et l'analyse de code malveillant

Directeur de thèse : **Davide Balzarotti**

Jury

Edgar Weippl,	Associate Professor, SBA Research and Technische Universität Wien	Reporter
Levente Buttyán,	Associate Professor, CrySyS Lab, Budapest University of Technology and Economics	Reporter
Juan Caballero,	Assistant Research Professor, IMDEA Software Institute	Examiner
Brendan Dolan-Gavitt,	Assistant Professor, New York University	Examiner
Bruno Martin,	Professor, Université Nice Sophia-Antipolis	Examiner
Pietro Michiardi,	Professor, Eurecom	Examiner
Refik Molva,	Professor, Eurecom	Examiner

T H È S E

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2015-ENST-0059

EDITE - ED 130

ParisTech Ph.D.

Ph.D. Thesis

to obtain the degree of Doctor of Philosophy issued by

TELECOM ParisTech

Specialisation in Computer Science and Networking

Publicly presented and discussed by

Mariano Graziano

September 28th, 2015

Advances in Modern Malware and Memory Analysis

Advisor : **Davide Balzarotti**

Committee in charge

Edgar Weippl,	Associate Professor, SBA Research and Technische Universität Wien	Reporter
Levente Buttyán,	Associate Professor, CrySyS Lab, Budapest University of Technology and Economics	Reporter
Juan Caballero,	Assistant Research Professor, IMDEA Software Institute	Examiner
Brendan Dolan-Gavitt,	Assistant Professor, New York University	Examiner
Bruno Martin,	Professor, Université Nice Sophia-Antipolis	Examiner
Pietro Michiardi,	Professor, Eurecom	Examiner
Refik Molva,	Professor, Eurecom	Examiner

THÈSE

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

Acknowledgments

I would like to acknowledge the following people, for their help and support during all the course of my PhD studies.

First, my advisor Davide Balzarotti. He gave me the possibility to join his group, and I moved to France without any idea about academic research. In these years, he has been always available to talk and discuss and I hope to be a good researcher in the future by following his advices. I have been very lucky to work with him. Thank you.

Second, all my “s3" collegues: Andrea, Andrei, Aurelien, Canali, Clementine, Davide, Giancarlo, Jelena, Jonas, Leyla, Luca, Matteo, Merve, Onur and Xiao. This experience has been great also because we are a great lab. Many times I was stuck and I will always remember the endless brainstorming sessions with Andrea and at the end we were able to find new solutions and create new papers.

Third, my friends in France, my own little Italy. Thanks Benza, British, Masche, Viotti and Alberto for all the dinners and beers together. My old friends in Italy, thanks Zame and Marco for your support, your messages and visits.

I am also really grateful to all the people of the “nops team". We had fun and we played a couple of nice CTF competitions. Thank you Maurizio, Luca, Kjell and Pecko. Another source of inspiration and of nice memories comes from the visitors students. Thank you Gabor, Xabier, Fabio and Flavio.

I also really thankful to Cisco for the internship and my period in USA. The VRT has been very welcoming and in particular I am really grateful to Alain and Shaun, they did much more than a manager and a colleague.

I would also like to thank my committee (professors Edgar Weippl, Levente Buttyán, Juan Caballero, Brendan Dolan-Gavitt, Bruno Martin, Pietro Michiardi, and Refik Molva) for agreeing to be reporters and examiners for my Ph.D. dissertation.

A very special thank you goes to Flavia for her love, advices and support.

Thanks finally to my parents (Giusi and Nicola) and my brother (Luca) for believing in me. Your unconditional support has been my strength and this thesis and all my publications are dedicated to you.

Contents

List of Figures

List of Tables

List of Publications

This thesis comprises four papers. The first three papers have been published to peer-reviewed academic conferences. The last one is currently under submission. The following list summarizes the aforementioned publications:

— Mariano Graziano, Corrado Leita and Davide Balzarotti
Towards network containment in malware analysis systems
28th Annual Computer Security Applications Conference (ACSAC 2012)
December 2012, Orlando, Florida, USA

— Mariano Graziano, Andrea Lanzi and Davide Balzarotti
Hypervisor Memory Forensics
Symposium on Research in Attacks, Intrusion, and Defenses (RAID 2013)
October 2012, Saint Lucia

— Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi and Davide Balzarotti
Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence
24rd USENIX Security Symposium (USENIX Security 2015)
August 2015, Washington DC, USA

— Mariano Graziano, Davide Balzarotti and Alain Zidouemba
ROPMEMU: A Framework for the Analysis of ROP Chains
Under Submission at NDSS 2016

Chapter 1

Introduction

It has been estimated that three billion people were connected to the Internet in 2015 [6], and this figure increases every year as entire regions in emerging markets are plugged to the cyberspace by telecommunication companies. The Internet, and in particular the World Wide Web (WWW), has simplified the life of millions of people and companies. Nowadays, many families have an Internet connection and own several devices such as computers, laptops, and smartphones that are able to connect to the network. As a consequence, in the last decade many business activities moved online and even governments foster institutions to move their services on the Web.

This process offers several advantages to end users. For instance, people can buy products online and make bank transfers from their living rooms, webcams help people to interact with their families overseas, and instant messaging programs allows for a free asynchronous communication. In general, these services reduce costs and save time to the final users.

Unfortunately, the Internet revolution and its transformations have also attracted criminal activities. Miscreants realized the lucrative business behind online services and recognized the role played by Internet as a fundamental pillar in modern economies. As a consequence, over 317 million new malware variants were discovered in 2014 [189]. In the last decade, malicious software has been developed by organized groups for financial gain. Their business is based on stealing credentials and information. In the case the machine does not contain valuable information, it can be rented to a third party and used to send spam or for distributed denial of service (DDoS) attacks.

More recently, malware and breaches have also been perpetrated by powerful governments and private corporations. A hidden war is fought with exploits and rootkits in order to exfiltrate information and gain an advantage against adversaries. In these cases, the enemy may range from groups of terrorists to legitimate nations, from private companies to dissidents. These silent attacks are called advanced persistent threats (APT) and are the core component of cyber-espionage campaigns.

The amount of money lost by private citizens and companies due to cyberattacks reached 400 billion dollars in 2014 [144]. This is a just a rough estimation and does not take into account damage to reputation, indirect costs, and compromised companies (like Sony [194] and Home Depot [197]) that do not publicly disclose their financial losses. Consequently, ordinary people and private companies in industrialized countries demand protection for their accounts and intellectual property. Security companies play an important and active role in this never ending war. They offer custom solutions to the private and public sector. For example, brand new computers are shipped with a pre-installed antivirus software, new start-ups promise to eradicate APTs, governments forced strict guidelines as well as certifications to guarantee a minimum level of security, specialized companies sell zero-day exploits and stealthy rootkits for lawful interceptions, and governments pay advanced trainings for their cyber army.

In some operations, security companies joined forces to take over botnets and arrest malware authors. Moreover, they created immense infrastructures to automatically collect and analyze the increasing number of suspicious samples. In fact, mainly due to packing and polymorphism, modern anti-malware companies collect an overwhelming number of new samples per day, for instance a well-known company like Virustotal [198] daily collects more than one million samples.

1.1 Modern Malware Analysis

Modern malware analysis is in large part automated, and only a small subset of the collected samples are manually analyzed by reverse engineering experts. In the last years, security companies deployed a complex infrastructure to collect samples from their customers and from ad-hoc vulnerable machines (honeypots). These infrastructures, often hosted on the cloud, analyze in real time the customers traffic, extract documents and executables, and analyze them inside an instrumented environment (normally called a *sandbox*). The system then applies several heuristics on the generated reports and an alarm is raised in case the file is considered malicious.

Unfortunately, this process has several limitations that may be exploited by advanced malware. For instance, samples may be designed to detect the instrumented environment and hide their real malicious behavior or they may be programmed to work only on a specific target machine. For this reason, the analysis of sophisticated malware often involves runtime information collected on the infected systems, typically in the form of a dump of the physical memory. In fact, from the memory of the infected machine it is possible to extract important artifacts and collect additional information while the malware is operating in its target environment. The combination of these two approaches is summarized in Figure 8.1. The analyst leverages both approaches, binary dynamic analysis and memory analysis, to have a broader view about a specific threat.

This dissertation proposes improvements to the modern malware and memory analysis. Although these fields have been extensively studied from different

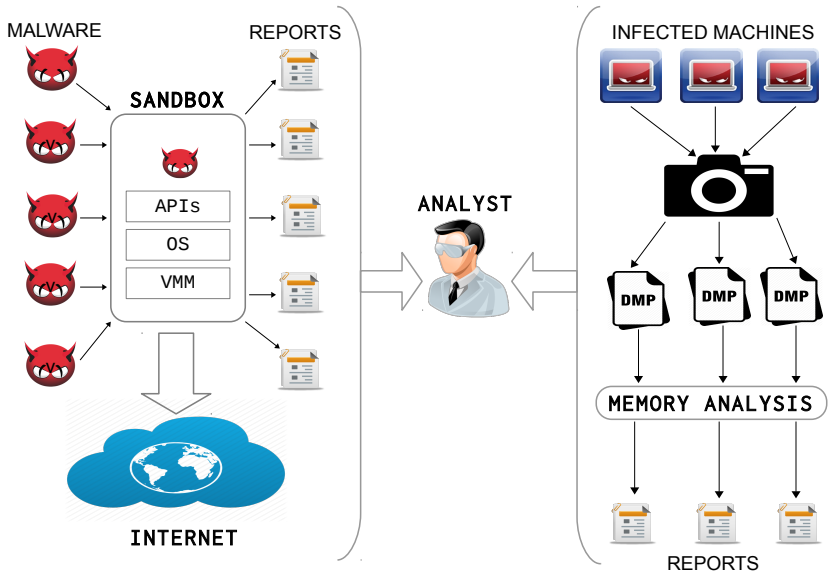

Figure 1.1 – Thesis overview

perspectives in the last years, there are still several aspects that may be significantly improved. In particular, sandboxes can be optimized to have more granular network containment techniques. In addition, researchers can monitor submitted executables to spot active malware developments on these online systems and prioritize the samples assigned for a manual analysis.

Along the same lines, memory analysis is still a young field with room for improvements. In this case, we proposed the first framework able to analyze virtual machines and hypervisors also when nested configurations are in place. Moreover, we leveraged memory analysis to cope with advanced threats that do not require code injection techniques.

1.2 Sandboxing Technology

Sandbox solutions are a key component of modern malware analysis as the overwhelming number of samples collected per day makes other solutions impractical. For instance, manual analysis does not scale and requires experts to dissect malware binaries. Second, an automated system is required to filter out irrelevant samples and collect valuable information in a reasonable amount of time. Third, static analysis techniques are often not effective against malicious files. To tackle these issues, security researchers devised a number of sandbox environments. These instrumented environments can be run in parallel and can be customized per sample. For instance, it is possible to run the same sample on both

Windows XP and Windows 8. Moreover, analysts can plug additional plugins to extend sandbox functionality (e.g user activity simulation).

1.2.1 Sandbox Design

Sandboxes are designed to collect malware behavior. Unfortunately, malware samples are evasive and very complex to analyze. Researchers have to isolate and actively monitor the relevant events such as the network traffic, the Windows registry, filesystem modifications, the creation of new processes, and suspicous memory operations. In order to gather this information, researchers can decide to deploy an in-guest monitor agent that collects the information by using hooking techniques. This agent can work both as a kernel- and user-land component. The kernel monitoring is necessary at least in case of kernel rootkit analysis. This malware category works directly at ring 0 and tampers the core of the operating system. Another approach is to deploy the monitor agent out-of-guest. In this case, the agent is implemented inside the hypervisor (or emulator) and virtual machine introspection techniques are used. Despite the fact that dynamic malware analysis has several advantages, in the last years malware authors have introduced anti-sandbox functionality to hinder the analysis. For this reason, security researchers have to carefully implement instrumented environments to be as stealthy as possible. Sandboxes can be built on top of either hypervisors or emulators. Both approaches have pros and cons and strive for the same result.

Full virtualization solutions instrument hypervisor code. In this way, researchers can extend the virtual machine monitor functionality and introduce the necessary modules to log the malware activity. The monitor component is outside the guest operating system, thus it should be impossible for the malware to detect the instrumentation code. Moreover, the majority of the instructions are executed directly on the CPU, the additional overhead is introduced only to monitor particular events of interest. In these cases, the hypervisor traps and executes the proper routine to record information. Unfortunately, this phase is not trivial and it is complicated by the so-called *semantic gap*. At this point, the hypervisor has to analyze the physical memory to rebuild the data structures of interest of the running guest operating system. This is really challenging and requires a deep knowledge about the operating system internals. For example, Windows systems have three different views about the process: `EPROCESS`, `KPROCESS` and `PEB`. The first two data structures keep track of vital information for the executive and the kernel subsystems while the last one represents the process in user-land. Moreover, the hypervisor does not have any state information so the mechanism to distinguish processes is by using the `CR3` register on `x86` systems. This register contains a physical address and points directly to the base address of the the first data structure implementing the memory management unit (MMU). Any event is associated to a process by inspecting this register and additional information is retrieved through corresponding data structures (e.g., `EPROCESS`). However, it is not trivial to locate and follow these data structures. The system needs heuristics to find the structures of interest in

the physical memory and then implement the translation mechanism (from virtual to physical addresses) to follow interesting pointers. More in general, researchers refer to this set of techniques as virtual machine introspection (VMI).

Commonly, virtualization is used as underlying technology to boot guest operating systems. In these cases, the hypervisor code is not instrumented and the monitor component is implanted inside the virtual environment. In this scenario, this component can be either a kernel driver or a dynamic library (DLL) designed to log functions of interest as well as the network activity. This option needs special precautions. By default, brand new virtual machines installed on top of the most common virtual machine monitors such as VMWare, Virtualbox, Xen, KVM contain many evidences about the virtual environment as well as about the underlying hypervisor. Researchers have to configure these machines and remove trivial detection points.

Emulation is a technology able to simulate assembly instructions via software. As a result, emulators can simulate complex programs such as operating systems. This approach is flexible, the emulator can implement assembly instructions of several architectures. In this way, it is possible to observe programs running on ARM on top of `x86` systems. Emulators provide easy-to-use instrumented environments. In particular, solutions like Qemu may be extended by researchers to log malware activity. The most common emulation approaches are: OS emulation and full system emulation.

OS emulation tries to emulate via software the behavior of the operating system. More in general, this approach is able to provide the result of common functions. In malware analysis the operating system in place on sandboxes is Windows (ranging from XP to the latest stable version) – the operating system most affected by malware. These OS releases are generally deployed with different service packs, some malware can show their real nature only in a very specific environment. In this configuration, researchers have to decide the functions of interest. Generally speaking, they decide to monitor suspicious calls, functions like `LoadLibrary`, `CreateRemoteThread`, `WriteProcessMemory`, etc are properly emulated. On Windows systems, there are two function families: Win32 APIs and native functions. Unfortunately, only Win32 APIs are well documented and considered stable. On the contrary, native functions have no documentation and can be changed at any time. Sandbox developers support the most common Win32 APIs, but these functions are a wrapper around the native ones that can talk directly to the kernel. Malware authors know this limitation and evade the analysis based on OS emulation by invoking native APIs.

Full system emulation is a technique able to emulate the entire operating system, this is possible by supporting hardware peripherals. In this configuration, the emulator is able to collect every instruction executed by the malware inside a target operating system. The monitor component can track all the memory read and write operations. All this information is really useful during a detailed analysis. In addition, the report have more insights and, in general, it is easier for an analyst to

figure out the nature of the sample. In this configuration, the more instructions the emulator supports, the more accurate and stealthy is the analysis.

Emulation and virtualization approaches are flexible and easy to deploy. Security researchers set up the machine, take a snapshot and can run thousands of samples on the same guest. Once the analysis is over, the snapshot is restored and the system is clean again. Despite the limitation and the possibile evasion techniques, these two solutions provide a good trade-off. In very specific cases, the analyst can decide to run the sample on a bare metal system. The monitor component can be directly installed on the host operating system. In this case, the analysis is accurate, there are no virtual components. Unfortunately, this approach does not scale. Once the machine is infected, the researcher has to install again the entire operating system.

1.2.2 Problem statement

Dynamic analysis is a powerful approach to uncover malware behavior, and sandboxes are the most common instance of this technique. These instrumented and virtual environments can run untrusted code in an isolated environment and can provide the analyst a very flexible and customizable analysis framework. Unfortunately, current sandboxes still suffer from several limitations. In this dissertation, we focus on two problems in the area of dynamic malware analysis.

First, malware analysis is not *repeatible*. In particular, the malware behavior often depends on the network context. This means that many samples interact with online servers and if these servers are not available the behavior (and therefore the analysis report) is affected. Moreover, some samples are only designed to run in specific target environments and would fail when executed elsewhere. Unfortunately, repeatability is a very important aspect of malware analysis and it is desirable in many scenarios. For example, researchers may want to re-analyze the same sample after months with a new technique, in order to gather more information. In a completely automated infrastructure based on parallel sandboxes this limitation may hinder and pollute the analysis and the reports.

Second, packing and polymorphism have become very common in malware and nowadays it is common to have many different samples for the same family. As a result, sandboxes are overloaded by binaries that are all equivalent from a behavior point of view. This phenomenon complicates the job of security analysts. In particular, the task of *distinguishing* new and important malware from the background noise of polymorphic and uninteresting samples is a very challenging open problem in the field.

1.3 Memory Analysis

Memory analysis comprises a set of techniques to analyze the content of system memory (RAM). In the last decade, it has gained popularity and it is now an

important step in many real investigations. Researchers proposed stable techniques to inspect physical memory, locate data structures of interest, and extract the necessary information. The popularity of this approach resides in the central role of the memory in a system. In addition, advanced attacks now exist that are located only in memory and do not leave any footprint inside the filesystem.

Memory analysis is an active research field that has rapidly evolved over the years. It can be performed both offline (memory forensics) and in an online fashion – but the two approaches typically adopt the same techniques. Memory forensics is based on the analysis of physical memory dumps, collected by acquisition tools and devices. In contrast, online analysis systems inspect the system memory live. This is possible by using programs able to export a special device that enables direct access to the physical memory.

Practitioners have to cope with several challenges. The semantic gap is a common issue. The information is stored in memory as a raw stream of bytes and experts need a deep knowledge about the operating system internals to extract and reconstruct the required artifacts. There are several available tools (such as Volatility, Rekall, and Memorize) that are designed to cope with this problem. They all start by locating important data structures. These data structures can reside in the physical memory at a fixed offset. Unfortunately, the increasing adoption of ASLR techniques in user- and kernel-space makes this approach less effective against the latest OS releases. A more reliable approach is based on walking through a number of intermediate data structures (starting from global symbols) in order to reach the target data. Finally, it is possible to create strong signatures to scan linearly the physical memory and discover all occurrences of a particular object. This phase is further complicated by the OS diversity, since the memory layout of data structures is not constant over different operating system releases. Therefore, the analysts need a profile in which every data structure is described in details.

Another common problem of memory analysis is address translation. Once the location phase is terminated, the analyst has an object containing several fields. However, any pointer is a virtual addresses and the memory analysis framework works only with the physical memory. Particularly, the framework needs to implement its own memory management unit (MMU). This implies the knowledge of the architecture, generally contained in the profile. Fortunately, the available tools are able to address all these challenges.

Besides these known and solvable problems, memory analysis, as a complementary approach, provides an unique point of view. This new perspective speeds up considerably the analysis time. For example, analysts can immediately isolate hidden processes. Volatility plugins like `psxview` compare the output of six different techniques to list the running processes. In this way, analysts can easily spot malicious processes. Moreover, in the last years, static analysis showed its weaknesses. Specifically, attackers can obfuscate their code and significantly hinder the analysis. The evolution of these techniques made static analysis almost ineffective. Consequently, researchers adopted new approaches. In particular, memory forensics may offer new insights for the analyst and, in most of the simple cases,

is able to defeat light forms of obfuscation such as packing. Another common use case is the deep analysis of an infection. For instance, malware commonly injects code and even entire DLLs into another process address space, this is known as code injection. Memory forensics offers approaches to automatically detect these threats (e.g., `malfind` Volatility plugin).

1.3.1 Problem statement

Memory analysis is a complementary approach in modern malware analysis. It is a rapidly growing area that has proven to be useful in many investigations – but it has still several limitations. In particular, in this dissertation we explore how memory analysis techniques can be extended to study two forms of advanced threats.

First, memory forensics is currently unable to detect and cope with any form of virtual machine monitor. Consequently, all the available tools cannot detect and transparently introspect guest operating systems. The situation is even worse in presence of nested configurations as the malware analyst has no tool to detect and dissect malicious hypervisors. In fact, sandboxes do not support nested virtualization and, to the best of our knowledge, there are no tools and techniques to monitor these possible advanced threats.

Second, the current approaches in memory forensics aim at finding intrusion evidences in the physical memory dumps. Commonly, these evidences involve artifacts that have been created or injected in memory by the malicious components. Volatility plugins like `psxview` and `malfind` are good example of tools that perform this task. Unfortunately, there is now an emerging trend of advanced threats that adopt code reuse techniques (such as return oriented programming) as a mean of obfuscation, to perform malicious computation without injected code. In these cases, both memory and binary analysis tools are completely ineffective to locate and dissect instances of code reuse, thus leaving the analyst blind to this new type of threats.

1.3.2 Contributions

In this thesis we propose a number of techniques to address unsolved problems in the areas of modern malware and memory analysis. In particular, the research presented in this document makes four individual contributions: two on the area of dynamic malware analysis, and two to improve memory forensics to support the analysis of advanced threats. Figure 8.2 shows the four contributions and how they are located in respect to the general picture.

Overall, we made the following contributions:

1. In Chapter 3 we present a technique to process millions of malware sample submissions received by a malware analysis sandbox and we propose a novel methodology to automatically identify *malware developments*. Our

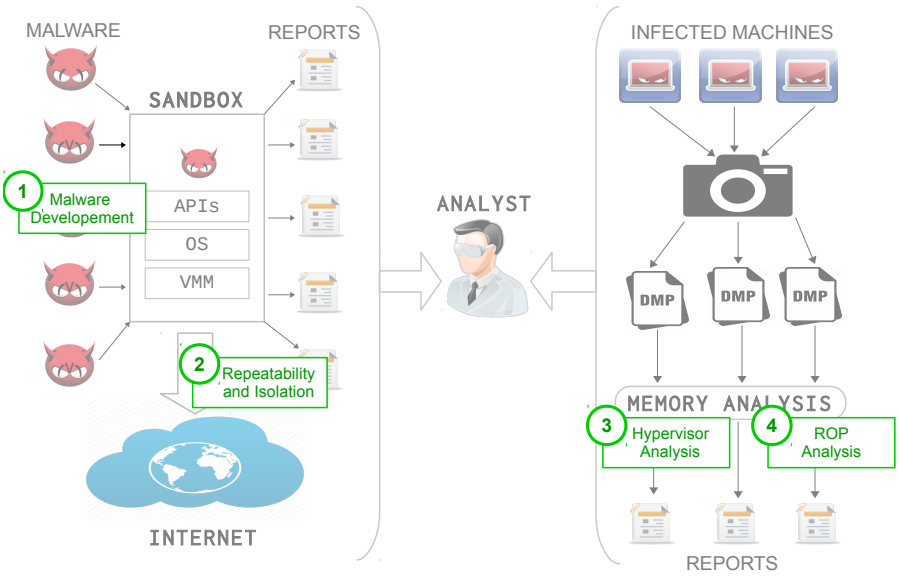

Figure 1.2 – Thesis Contributions

approach is based on the combination of static and dynamic analysis and file submission features. We also use data mining and machine learning techniques to acquire more insights about the dynamics of malware development.

2. In Chapter 4 we investigate the use of protocol learning techniques to model the traffic generated during the execution of malware samples in order to automatically reply malware conversations. Using this technique, we developed a novel network containment system and we showed that, even with some limitations, it is possible to achieve *full containment* and perform a *repeatible* analysis, also in cases in which the malware behavior depends on external hosts like C&C servers.

3. In Chapter 5, we proposed the first memory forensic framework to analyze hypervisor structures in physical memory dumps. In addition, we analyzed nested configurations and developed a transparent mechanism to recognize and support the address space of virtual machines. Our approach allows to perform memory forensics to analyze malicious hypervisors, as well as compromised virtual machines that are part of large virtualized environments.

4. In Chapter 6 we present a set of techniques to perform binary and memory analysis of sophisticated attacks that do not rely on any injected code.

In particular, our approach identifies and discusses the main challenges that complicate reverse engineering code implemented using return oriented programming (ROP). In addition, we propose an emulation-based framework to dissect, reconstruct, and simplify ROP chains directly from a physical memory dump. We tested our tool with the most complex example proposed so far: a rootkit made of several ROP chains, with a total of 215,913 gadgets.

Chapter 2

Related Work

Malware and memory analysis have been extensively studied in the literature. Unfortunately, the industry is still struggling with many aspects of the analysis of malicious software. Specifically, malware has evolved over the years and we witnessed its evolution from a niche problem to a plague for our daily lives. In this endless war, researchers try to chase malware writers and protect the users.

The first malicious programs were simple and designed to be a technical exercise. Smart and bored teenegers coded malicious programs to show their skills to the world. In their creations there was not any profit intent. In some cases, the payload was a text message. In other cases, the goal was cybervandalism. Ther infection vectors were based on floppy disks and only later on the network. As a consequence of these first virus samples, the first antivirus companies appeared in 1987. The first engines were based on the concept of *signature*. Immediately, the VX communities (virus writers groups) adapted their techniques and easily bypassed this new countermeasure. It was 1989 when polymorphism appeared for the first time [34] and it was the beginning of an arms race still in progress today.

The Internet revolution brought many users online. As a side effect, many cyber-criminal groups appeared. In these years, the Internet changed considerably. The cyberspace was no more a place for few and smart people and the underground spirit faded away due to the lucrative business created by large corporations. On the other side, miscreants realized malicious software might be used as a new mean to make money. As a consequence, security companies had to evolve to fight organized and skilled groups of cyber-experts.

In the last decade, researchers proposed several techniques to make malware analysis more efficient and effective. The first approaches were based on manual analysis and, consequently, many advances have been proposed on static and program analysis. Nevertheless, malware authors devised advanced forms of obfuscation to hinder the manual analysis, and adopted polymorphism and metamorphism to bypass naive signature approaches. As a consequence, security companies invested resources on dynamic analysis and, as expected, miscreants started intro-

ducing countermeasures to avoid the exectution on instrumented environments – in an ongoing cat-and-mouse game.

The work in this thesis covers modern automated malware analysis. This approach is based on two technologies: sandboxes and memory analysis. In this chapter we will summarize the main contributions to these fields. In particular, in section 2.1 we will introduce related work on malware analysis and sandbox technologies. In section 2.2 we will overview memory analysis.

2.1 Dynamic Malware Analysis

Dynamic malware analysis executes the sample and observes its behavior at run-time. Both the logging and the analysis process may be achieved in several ways and at different layers. Moreover, the instrumentation environment heavily depends on the underneath operating system. For these reasons, over the years, researchers proposed many environments leveraging different technologies. This versatility and room for further customizations made sandboxes the most common instance of dynamic analysis. In the last years, researchers significantly improved these systems and, nowadays, they are an important component used actively by security companies to combat malware.

This technology considerably evolved over the years. The first rudimentary approaches logged only a subset of the events of interest. TTAnalyze [41] is the first comprehensive framework to analyze malicious samples in a controlled environment. The evolution of this project is Anubis [19], the first public online sandbox. Nowadays, there are several sandboxes worth mentioning. Some of them are freely available online such as Malwr [25], ThreatExpert [26] and Anubis [19]. Others are open source and can be deployed internally such as Cuckoo [23] and Zerowine [107]. Other sandboxes (e.g., Joebox [105], Fireeye [80], Bromium [50] and Lastline [122]) are proprietary and a customer can have both an online access and a private instance. All these solutions always provide a detailed report to the analyst but the underlying technology and implementation may differ. The first versions of these malware analysis systems supported only user-land threats and the logging engine was implemented inside the guest operating system. Moreover, the logging capability was simply a system calls/APIs tracer. Some sandboxes (for example CWSandbox [20] and Cuckoo [23]) use their own hooking library while others prefer to leverage existing systems like Detours [145]. Successively, security experts refined the logging component to collect more information and show a more accurate report. The kernel mode support has been added in a second phase. Although, the number of ring0 samples is considerably smaller than the number of user-land malware, kernel support is necessary to have an immediate idea about the behavior of complex kernel rootkits. In the third phase of sandbox platforms, the researchers coped with the intrumentation stealth. The widespread adoption of all the aforementioned specific precautions forced the miscreants to introduce

anti-sandbox routines in the malware sample. In this way, malicious programs do not disclose their behavior and remain unnoticed.

These anti-sandbox functionalities are designed to detect the virtual environment and the software underneath (tipically either a hypervisor or an emulator). Specifically, the virtual environment may contain many evidences. For instance, Windows operating systems on top of Virtualbox, a popular virtual machine monitor, can be easily detected by looking at the Windows Guest VirtualBox devices (`\\Device\\VBoxGuest`) or at the MAC address ranges. In addition, the Windows registry is another source of evidences. Many keys contain VirtualBox strings. Although the number of these possible checks may be endless, researchers can easily patch the vast majority of them. However, public and online sandboxes have to cope with other simple detection points. In particular, the instrumented environment has to be randomized otherwise miscreants can easily detect the sandbox. Avtracker [3] shows this problem and provides information to easily detect public online sandboxes. The author of the website periodically interacts with the online services and collect possible detection points such as the public IP, the user and computer name.

Cybercriminals may exploit also other detection points. These points may be more problematic to patch and fix and reside in defects of the underlying software. Even worse, a small percentage is intrinsic and shows the limit of the technology in use (virtualization or emulation). For example, the *timing attacks* exploit these intrinsic limitations [39, 58, 68]. In this situation, a malware author can execute the same instruction in an emulated environment and in a physical machine. As a result, she would obtain two different timestamps. After a testing phase, she can introduce the checking routine on her malware and detects the time discrepancy using the `rdtsc` assembly instruction on `x86` machines. This trick has been adopted by several malware families and it is often observed in the wild.

The *emulator bugs* show the limitations of software approaches. For instance, the Intel instructions set is complex and contains thousands of instructions. As a result, the software implementation of these instructions may contain bugs. Moreover, the emulator authors may decide to implement only a subset of the instructions and ignore uncommon side effects. Therefore, in some cases, it is possible that the execution of an assembly instruction on a virtual CPU may behave differently compared to a real one and this discrepancy can be used to detect the virtual environment. In addition to exploiting software bugs, attackers may leverage undocumented opcodes to complicate the analysis. Consequently, the emulator disassembler may fail to decode the opcodes to a valid assembly instruction. Paleari et al. [157] have developed an automated framework to detect these defects. In particular, they have studied the implementation of the CPU in Qemu and Bochs to build a set of reliable red-pills. The authors discovered 20,728 red-pills for detecting Qemu and 2973 for detecting and Bochs. In this set of red-pills there is also the original Rutkowska's pill [168]. It is important to note that this technique is generic

and can be applied to CPU virtualizer [143] as well as to other architectures. Again, these tricks have been already observed in the wild.

Consequently, researchers removed the most common evidences of the virtual environment and moved the logging technology in the hypervisors (or emulator) to overcome any possible detection. Initial works, such as the one proposed by Liston et al. [137], focused on removing specific artifacts in VMWare that are targeted by well-known checks. Successively, practitioners moved the implants out of the guest operating system. The first work in this direction and strictly related to dynamic malware analysis is VMwatcher [102]. Ether [69] is the first successful transparent instrumented analysis system and theoretically addressed all the detection points. However, Pek et al. [161] found implementation bugs and managed to detect the virtual environment. Although Ether and similar systems are successful in hiding their presence, they inevitably incur a performance penalty that is prohibitive for the deployment on real large-scale automated malware analysis environments. V2E [204] and DRAKVUF [129] aim at the ideal transparency and optimal performance. Specifically, V2E combines transparency and efficiency for an in-depth analysis. The tool comprises two phases: record and replay. The first is based on KVM and is transparent while the second one is based on TEMU [46] for further inspections. DRAKVUF solves technical challenges for the out-of-the-box support also for kernel rootkits and leverages the advances in the virtualization techology (e.g., Extended Page Tables) to have a low analysis overhead. Other system-wide instrumentation frameworks similar to V2E and DRAKVUF are built on top of emulators and a common choice is Qemu. The first comprehensive framework is TEMU [46] from the BitBlaze team and its new and enhanced version DECAF [91]. S2E [59] provides powerful symbolic execution functionality as well as a component to translate the Qemu IR (TCG) to LLVM bitcode. Finally, Panda [74] combines features from the aforementioned approaches to ease reverse engineering tasks. Moreover, it specifically focuses on the repeatability of dynamic analysis and on the modularity of the framework, easily extendible through plugins.

Although dynamic analysis is a powerful weapon and a pillar in modern malware analysis, it is not perfect and can be improved considerably. In this dissertation, we propose two advances to dynamic malware analysis. The first is about network containment in order to achieve a repeatable analysis. The second propose a set of techniques to monitor the samples submitted to a sandbox to discover possible malware developments. For this reason, the remaining part of this section focus on these two areas.

2.1.1 Network Containment

Several different strategies have been proposed to address the problem of network containmnent and the quality of the dynamic analysis. In particular, the concept of quality refers to both the need to allow connectivity to external hosts (to ex-

Approach	**Containment**	**Quality**
Full Internet access	$\times$	$\sim$
Filter/redirect specific ports	$\sim$	$\sim$
Common service emulation	$\surd$	$\sim$
Full isolation	$\surd$	$\times$

Table 2.1 – Network access strategies in dynamic analysis

pose the malware interesting behavior) and to the need to make the analysis process repeatable. Table 8.1 summarizes the previous work in four different categories.

Full Internet access. The most straightforward approach consists in providing the sandbox with full Internet access. A similar approach is however unacceptable from a containment standpoint: the malware sample is left free to propagate to victims, or to participate into other types of malicious activities (e.g., DoS, spam). The quality of the analysis is also only partially acceptable: the sample is left free to interact with external hosts upon execution, but its behavior becomes dependent on the state of external hosts, leading to the problems underlined in [125].

Filter/redirect specific ports. The containment problem associated to Full Internet access is rarely discussed in Internet-connected sandboxes such as Anubis [19], CWSandbox [20] and others. From informal discussions with the maintainers, it appears to be common practice for the public deployment of these sandboxes to employ simple filtering or redirection rules, in which TCP ports commonly associated to malicious scans (e.g. port 139 and port 445) are either blocked or redirected towards honeypots. This partially solves the containment problem: SMB vulnerabilities are a very common propagation vector for self-propagating malware, that can be easily prevented with such measures. However, this approach is not able to deal with other types of activity whose nature cannot be easily discerned from the TCP destination port. A similar attempt to perform containment through redirection was implemented also in the context of honeyfarms such as Potemkin [200] and GQ [64]. In such deployments, the authors investigated the idea of reflecting outbound traffic generated by infected virtual instances of the honeyfarm towards other instances of the same honeyfarm. A similar approach proved to be valuable for the analysis of malware propagation strategies, but was not effective at dealing with other types of traffic such as C&C communication. In fact, redirecting a C&C connection attempt towards a generic honeyfarm virtual machine is not likely to generate meaningful results. Kreibich et al. [117] have recently improved GQ making it a real and versatile malware farm. They have addressed the containment problem with precise policies but their approach has not dealt with the repeatability issue.

Common service emulation. Sandboxes such as Norman Sandbox prevent the executed malware from connecting to the Internet, and provide instead generic service implementations for common protocols such as HTTP, FTP, SMTP, DNS and IRC. A similar approach was revisited and enhanced by Ionue et al. in [94], a two-pass malware analysis technique in which the malware sample is allowed to interact with a "miniature network" generated by an Internet emulator able to provide a variety of dummy services to the executed malware sample. All these approaches are however limited, and rely on a-priori knowledge of the communication protocols employed by the malware sample. Malware often uses variations of standard protocols, or completely ad-hoc communication protocols that cannot be handled through dummy services. Yoshioka et al. [205] have tried to tackle this problem by incrementally refining the containment rules according to the dynamic analysis results. While such an approach provided an elegant solution to the containment problem, it did not address the quality of the analysis and it did not attempt to remove dependencies between the malware behavior and the state of the external Internet hosts involved in the analysis.

Full isolation. Completely preventing the malware sample from interacting with Internent hosts ensures a perfect containment of its malicious activity. However, the complete inability to interact with C&C servers and repositories of additional components is likely to severely bias the outcomes of the dynamic analysis process.

Table 8.1 underlines a partial trade-off between the containment problem and that of ensuring the quality and repeatability of the analysis. On the one hand, running the malware sample in full emulation addresses all the containment concerns but, by barring the malware sample from communicating with the external hosts it depends on, it strongly biases the results of the dynamic analysis (i.e., the sample may only go as far as trying to connect to the hosts but without exposing any real malicious behavior). On the other hand, providing the sandbox with full Internet connectivity increases the analysis quality but it does not solve the repeatability problem, and it also raises important ethical and legal concerns.

In chapter 4 we address this problem by exploring the use of protocol learning techniques to automatically create network interaction models for the hosts the malware depends on upon execution (even in presence of custom and undocumented protocols), and using such models to provide the sandbox with an isolated, yet rich network environment.

2.1.2 Malware Development

While there has been an extensive amount of research on malware analysis and detection, very few works in the literature have studied the datasets collected by public malware dynamic analysis sandboxes. The most comprehensive study in

this direction was conducted by Bayer et al. [40]. The authors looked at two years of Anubis [19] reports and they provided several statistics about malware evolution and about the prevalent types of malicious behaviors observed in their dataset.

Lindorfer et al. [135] conducted the first study in the area of malware development by studying the evolution over time of eleven known malware families. In particular, the authors documented the malware updating process and the changes in the code for a number of different versions of each family. In our study we look at the malware development process from a different angle. Instead of studying different versions of the same well known malware, in chapter 3 we propose a large-scale detection of the authors of the malware at the moment in which they interact with the sandbox itself. In a different paper, Lindorfer et al. [136] proposed a technique to detect environment sensitive malware. The idea is to execute each malware sample multiple times on several sandboxes equipped with different monitoring implementations and then compare the normalized reports to detect behavior discrepancies.

A similar research area studies the phylogeny [89] of malware by using approaches taken from the biology field. Even if partially related to our contribution, in our study we were not interested in understanding the relationship between different species of malware, but only to detect suspicious submissions that may be part of a malware development activity.

In a paper closer to our work, Jang et al. [101] studied how to infer the software evolution looking at program binaries. In particular, the authors used both static and dynamic analysis features to recover the software lineage. While Jang's paper focused mostly on benign programs, some experiments were also conducted on 114 malicious software with known lineage extracted from the Cyber Genome Project [24]. Compared to our work, the authors used a smaller set of static and dynamic features especially designed to infer the software lineage (e.g., the fact that a linear development is characterized by a monotonically increasing file size). Instead, we use a richer set of features to be able to distinguish malware developments from variations of the same samples collected on the wild and not submitted by the author. While our approaches share some similarities, the goals are clearly different.

Other approaches have been proposed in the literature to detect similarities among binaries. Flake [81] proposed a technique to analyze binaries as graphs of graphs, and we have been inspired by his work for the *control flow analysis* described in chapter 3. Kruegel et al. [118] proposed a similar technique in which they analyzed the control flow graphs of a number of worms and they used a graph coloring technique to cope with the graph-isomorphism problem.

Finally, one step of our technique required to cluster together similar malware samples. There are several papers in the area of malware clustering [92, 97, 100, 201]. However, their goal is to cluster together samples belonging to the same malware family as fast as possible and with the highest accuracy. This is a crucial task for all the Antivirus companies. However, our goal is different as we are interested in clustering samples based only on binary similarity and we do not

have any interest in clustering together members of the same family based on their behavior.

2.2 Memory Analysis

In the last decade, digital investigations have considerably evolved. Researchers and practitioners have proposed efficient and effective methodologies to make digital forensics scientifically comparable to the traditional forensics in use by law enforcement. An important role in this evolution is represented by the first Digital Forensic Research Workshop (DFRWS) in 2001 [158]. Thanks to DFRWS, academics and forensics experts joined forces to create a community and systematically study the field to propose tools and methodologies as much rigorous as possible. At the beginning, digital forensics assisted law enforcement investigations and, over the years, the collected evidences have been regulated and accepted by the courts. Moreover, digital forensics become an active field of research.

Memory forensics is a branch of digital forensics and has been studied extensively since 2004 when Carrier et al. [56] proposed Tribble, a PCI based acquisition device for physical memory. In 2005, the DFRWS launched a challenge on memory analysis. The challenge comprised a physical memory dump from a compromised Windows operating systems and several questions about the breach. To answer the questions, researchers had to create new tools and technique to analyze and extract information from the memory dump. The goal of the organizers was to motivate researchers to investigate and improve this fascinating research area. In the same year, Movall et al. [150] discussed a suite for the analysis of Linux physical memory. In 2006, Petroni et al. [162] presented FATkit, a modular framework for the inspection of the physical memory. FATKit supports Linux and Windows operating systems, the reconstruction of the address space (e.g., IA-32) and has been developed following the Carrier's approach on the abstraction layers [55]. The evolution of FATkit is Volatility [12], currently the de-facto open source memory forensics framework. Before Volatility and its predecessor FATkit, many researchers released their own custom tools and techniques to extract single artifacts (e.g., the process list). This is the case of PTfinder from Schuster [175], Stewart's pmodump [186], Kornblum's studies [115, 116] and Dolan-Gavitt's publications [72, 73], just to name a few. Along the same line, researchers proposed dumping tools [187, 188] to ease the acquistion of the physical memory for different operating systems. In 2008, the DFRWS launched another memory analysis challenge. This time the focus was on the creation of methodologies and tools for the Linux operating system [60]. Again, the organizers goal was to foster researchers and improve the field. Similarly, in 2010 SSTIC challenged the french community to create tools for the analysis of the physical memory of a device run-

ning Android [10, 82]. Successively, researchers continued to improve the memory forensic field and added support for MAC OS X [12, 123] and FreeBSD [123].

In addition to open source solutions, many companies created their closed-source memory forensics framework. This is the case for Mandiant (now Fireeye) with Memoryze [141] and HBGary with Responder Professional [90]. This phenomenon shows the interest of the private sector on memory forensics. Unfortunately, at the moment, all the available frameworks can be easily defeated. These weakeness have been already documented by academics [165] and indipendent researchers [98, 140, 180] but the developers of the memory forensic frameworks did not address these critical issues so far. More recently, Case et al. [84] analyzed the new compressed RAM and extensively studied the swap files on Linux and MAC OS X. Similarly, Cohen implemented and adapted Kornblum's work [116] in Rekall [86], a spinoff of Volatility proposed by Google, for a correct and in-depth analysis of the Windows pagefile. In parallel, researchers have tested and analyzed the memory to extract many artifacts not necessarily related to the operating system components (e.g., processes, drivers and modules). For example, Alex Halderman et al. [88], described several attacks where they exploited DRAM remanence effects to recover cryptographic keys and other sensitive information. More recently, the so-called *cold boot attack* has been tested on Android [151] and its effectiveness has been confirmed while it did not work as expected [87] on DDR3 chips. Additionally, memory forensics have been used to discover malicious programs running unnoticed on the victim computer. For instance, Bianchi et al. [44] proposed *Blacksheep* to identify machines infected by a rootkit on a cloud infrastructure. The authors built a series of Volatility plugins to compare the snapshots of the different machines and implemented several heuristics to spot rootkit evidences. KOP [54] and MAS [66] apply memory analysis techniques on a single machine to locate malicious code running at kernel level but, unfortunately, they both require the source code of the operating system. More recently, MACE [79] extended KOP idea but using supervised learning techniques on pointers to build a kernel objects graph and detect kernel rootkits without access to the source code. Another interesting advance has been presented by Saltaformaggio et al. [173]. With DSCRETE, a system able to identify the information of interest in a memory dump and properly render its content by using its own application logic. In this way, the analyst does not need to know the memory layout of the data structures containing the information she seeks.

The 2015 DFRWS challenge focused again on memory forensics, this time on the analysis of GPU memory [4] in fact, researchers already proposed rootkits GPU-based [120] and observed in the wild malware authors that leveraged GPU to mine bitcoins [11]. Villani et al. [33] presented a detailed analysis of the GPU internals and described how a forensic examiners can cope with these threats. Along the same line, in the future, forensic analysts have to face advanced threats and create tools and techniques to dissect and analyze these new attacks. In this dis-

sertation, we will improve the field by adding the support to locate (potentially malicious) hypervisors and virtual machines on physical memory dumps. In addition, we allow the transparent offline introspection of the guest operating systems and detect nested configurations. In the literature, researchers have already proposed malicious hypervisors [70, 103, 111] that from an host operating system can take control of the entire machine. More recently, these threats have evolved and are able to undermine the targeted computer directly from the BIOS [146]. Unfortunately, up to the present, no memory forensics tools were able to cope with these treats. The second contribution proposed in this thesis aims at detecting modern and advanced attacks that do not inject any code in the victim operating system. This class of attacks is called *code reuse* attacks and have many instances such as ROP [178], JOP [48], BROP [47], SROP [49] and JIT-ROP [183]. In this thesis, we propose a framework based on memory analysis and emulation to analyze and dissect complex ROP payloads. We specifically focus on ROP because it is the most commong instance observed in the wild of code reuse attacks.

2.2.1 Hypervisors and Virtual Machines

Several papers proposed systems to search kernel and user-space memory structures in memory with different methodologies. Dolan-Gavitt et al. [75] presented a research work in which they automatically generated robust signatures for important operating system structures. Such signatures can then be used by forensic tools to find the objects in a physical memory dump.

Other works focused on the generation of strong signatures for structures in which there are no values invariant fields [130,133]. Even though these approaches are more general and they could be used for our algorithm, they produce a significant number of false positives. The approach we present in chapter 5 is more ad-hoc, in order to avoid false positives.

Another general approach was presented by Cozzie et al. in their system called Laika [62], a tool to discover unknown data structures in memory. Laika is based on probabilistic techniques, in particular on unsupervised Bayesian learning, and it was proved to be very effective for malware detection. Laika is interesting because it is able to infer the proper layout also for unknown structures. However, the drawback is related to its accuracy and the non negligible amount of false positives and false negatives. Lin et al. have developed DIMSUM [207] in which, given a set of physical pages and a structure definition, their tool is able to find the structure instances even if they have been unmapped.

Even though a lot of research have been done in the memory forensics field, to the best of our knowledge there is no previous works on automatic virtualization forensics. Our work is the first attempt to fill this gap.

Finally, it is important to note that several of the previously presented systems have been implemented as a plugin for Volatility [13] - the standard the facto for open source memory forensics. Due to the importance of Volatility, we also decided to implement our techniques as a series of different plugins and as a patch to

the main core of this framework.

2.2.2 Advanced Threats

Return Oriented Programming has been extensively studied in the scientific literature from several perspectives. However, very few works have presented novel techniques dedicated to the analysis of ROP chains and in this section we will focus only on those researches.

In this direction, the first study has been conducted by Lu et al. [139]. The authors proposed DeRop, a tool to convert ROP payloads into normal shellcodes, so that their analysis can be performed by common malware analysis tools. However, the authors tested the effectiveness of their system only against standard exploits containing really simple ROP chains. In chapter 6, we adopt some of the transformations proposed by DeRop – which we complement by a number of novel techniques required to deal with the large and complex chains of a ROP rootkits. Our main goal is also more ambitious, as we want to achieve a full code coverage of the ROP payload, also in the presence of dynamically generated chains.

In another paper similar to our work, Yadegari et al. [203] proposes a generic approach to deobfuscate code, in which the authors considers ROP as a form of obfuscation. Their system is based on bit-level taint analysis that is applied to existing execution traces and can be used to deobfuscate the control flow graph. In addition, the paper also adopts transformations similar to the ones proposed by DeRop to handle ROP payloads. Even though *Chuck* had already been released at the time, the authors claimed that no complex example of ROP chains was available, and they tested the system against small examples with a simple control flow logic. Moreover, the proposed system does not emulate the ROP chain and does not perform any code coverage. Instead, it focuses on the simplification of existing execution traces.

Another interesting research direction focused on the problem of locating ROP chains in memory and potentially profile their behavior [163, 185]. ROPMEMU can leverage these techniques to identify the persistent ROP chains. The profiling phase proposed in these papers were quite simple, and it may fail in presence of complex ROP chains. To overcome these limitations, we adopted an approach based on CPU and memory emulation. Finally, these techniques do not work in presence of packed ROP chains [138] or chains which are dynamically generated at runtime [199].

Up to today, all analysis and identification systems proposed in the literature have focused on simple user-space exploits. Therefore, the technique presented in chapter 6 is the only available solution that supports the analysis of a real kernel rootkit implemented in ROP.

Chapter 3

Malware Developments on Online Sandboxes

3.1 Introduction

In this chapter, we propose a novel methodology to automatically identify *malware development* cases from the samples submitted to a malware analysis sandbox. The results of our experiments show that, by combining dynamic and static analysis with features based on the file submission, it is possible to achieve a good accuracy in automatically identifying cases of *malware development*. Our goal is to raise awareness on this problem and on the importance of looking at these samples from an intelligence and threat prevention point of view.

Two important and distinct observations motivate our work. First, it is relatively common that malware samples used to carry out famous targeted attacks were collected by antivirus companies or public sandboxes long before the attacks were publicly discovered [45]. For instance, the binaries responsible for operation Aurora, Red October, Regin, and even some of the new one part of the Equation Group were submitted to the sandbox we used in our experiments several months before the respective attacks appeared in the news [22, 27, 108, 119, 147, 193]. The reasons behind this phenomenon are not always clear. It is possible that the files were automatically collected as part of an automated network or host-based protection system. Or maybe a security analyst noticed something anomalous on a computer and wanted to double-check if a suspicious file exhibited a potentially malicious behavior. It is even possible that the malware developers themselves submitted an early copy of their work to verify whether it triggered any alert on the sandbox system. Whatever the reason, the important point is that no one paid attention to those files until it was too late.

The second observation motivating our study is the constant arm race between the researchers that put continuous effort to randomize their analysis environments, and the criminals that try to fingerprint those systems to avoid being detected. As a consequence of this hidden battle, malware and packers often include evasion

techniques for popular sandboxes [28] and updated information about the internal sandbox details are regularly posted on public websites [3]. These examples prove that there must be a constant interaction between malware developers and popular public malware analysis services. This interaction is driven by the need to collect updated information as well as to make sure that new malware creation would go undetected. Even though detecting this interaction might be very difficult, we believe it would provide valuable information for malware triage.

Up to the present, malware analysis services have collected large volumes of data. This data has been used both to enhance analysis techniques [37, 148] and to extrapolate trends and statistics about the evolution of malware families [40]. Unfortunately, to the best of our knowledge, these datasets have never been used to systematically study malware development and support malware intelligence on a large scale. The only public exception is a research recently conducted by looking at VirusTotal to track the activity of specific high-profile hacking groups involved in APT campaigns [71, 206].

In this paper, we approach this objective by applying data-mining and machine learning techniques to study the data collected by Anubis Sandbox [19], a popular malware dynamic analysis service. At the time we performed our analysis, the dataset contained the analysis reports for over 30 millions unique samples. Our main goal is to automatically detect if miscreants submit their samples during the malware development phase and, if this is the case, to acquire more insights about the dynamics of malware development. By analyzing the metadata associated to the sample submissions, it might be possible to determine the software provenance and implement an early-warning system to flag suspicious submission behaviors.

It is important to understand that our objective is not to develop a full-fledged system, but instead to explore a new direction and to show that by combining metadata with static and dynamic features it is possible to successfully detect many examples of malware development submitted to public sandboxes. In fact, our simple prototype was able to automatically identify thousands of development cases, including botnets, keyloggers, backdoors, and over a thousand unique trojan applications.

3.2 Overview and Terminology

There are several reasons why criminals may want to interact with an online malware sandbox. It could be just for curiosity, in order to better understand the analysis environment and estimate its capabilities. Another reason could be to try to escape from the sandbox isolation to perform some malicious activity, such as scanning a network or attacking another machine. Finally, criminals may also want to submit samples for testing purposes, to make sure that a certain evasion technique works as expected in the sandbox environment, or that a certain malware prototype does not raise any alarm.

In this paper, we focus on the detection of what we call *malware development*. We use the term *"development"* in a broad sense, to include anything that is submitted by the author of the file itself. In many cases the author has access to the source code of the program – either because she wrote it herself or because she acquired it from someone else. However, this is not always the case, e.g., when the author of a sample uses a builder tool to automatically generate a binary according to a number of optional configurations (see Section 3.6 for a practical example of this scenario). Moreover, to keep things simple, we also use the word *"malware"* as a generic term to model any suspicious program. This definition includes traditional malicious samples, but also attack tools, packers, and small probes written with the only goal of exfiltrating information about the sandbox internals.

Our main goal is to automatically detect suspicious submissions that are likely related to malware development or to a misuse of the public sandbox. We also want to use the collected information for malware intelligence. In this context, *intelligence* means a process, supported by data analysis, that helps an analyst to infer the motivation, intent, and possibly the identity of the attacker.

Our analysis consists of five different phases. In the first phase, we filter out the samples that are not interesting for our analysis. Since the rest of the analysis is quite time-consuming, any sample that cannot be related to malware development or that we cannot process with our current prototype is discarded at this phase. In the second phase, we cluster the remaining samples based on their binary similarity. Samples in each cluster are then compared using a more fine-grained static analysis technique. Afterwards, we collect six sets of features, based respectively on static characteristics of the submitted files, on the results of the dynamic execution of the samples in the cluster, and on the metadata associated to the samples submissions. This features are finally provided to a classifier that we previously trained to identify the *malware development* clusters.

3.3 Data reduction

The first phase of our study has the objective of reducing the amount of data by filtering out all the samples that are not relevant for our analysis. We assume that a certain file could be a candidate for malware development only if two conditions are met. First, the sample must have been submitted to the public sandbox *before* it was observed in the wild. Second, it has to be part of a *manual* submission done by an individual user – and not, for example, originating from a batch submission of a security company or from an automated malware collection or protection system.

We started by filtering out the large number of batch submissions Anubis Sandbox receives from several researchers, security labs, companies, universities and registered users that regularly submit large bulks of binaries. As summarized in Table 1, with this step we managed to reduce the data from 32 million to around 6.6 million binaries. These samples have been collected by Anubis Sandbox from 2006 to 2013.

Dataset	Submissions
Initial Dataset	32,294,094
Submitted by regular users	6,660,022
Not already part of large submissions	522,699
Previously unknown by Symantec	420,750
Previously unknown by VirusTotal	214,321
Proper executable files	184,548
Final (not packed binaries)	121,856

Table 3.1 – Number of submissions present in our dataset at each data reduction step.

Then, to isolate the new files that were never observed in the wild, we applied a two-step approach. First, we removed those submissions that, while performed by single users, were already part of a previous batch submission. This reduced the size of the dataset to half a million samples. In the second step, we removed the files that were uploaded to the sandbox *after* they were observed by two very large external data sources: Symantec's Worldwide Intelligence Network (WINE), and VirusTotal.

After removing corrupted or not executable files (e.g, Linux binaries submitted to the Microsoft Windows sandbox), we remained with 184,548 files that match our initial definition of candidates for malware development. Before sending them to the following stages of our analysis, we applied one more filter to remove the packed applications. The rationale behind this choice is very simple. As explained in Section 3.4, the majority of our features work also on packed binaries, and, therefore, some potential malware development can be identified also in this category. However, it would be very hard for us to verify our results without having access to the decompiled code of the application. Therefore, in this paper we decided to focus on unpacked binaries, for which it is possible to double-check the findings of our system. The packed executables were identified by leveraging the SigBuster [112] signatures.

Table 1 summarizes the number of binaries that are filtered out after each step. The filtering phase reduced the data to be analyzed from over 32 millions to just above 121,000 candidate files, submitted by a total of 68,250 distinct IP addresses. In the rest of this section we describe in more details the nature and role of the Symantec and VirusTotal external sources.

Symantec Filter

Symantec Worldwide Intelligence Network Environment (WINE) is a platform that allows researchers to perform data intensive analysis on a wide range of cyber security relevant datasets, collected from over a hundred million hosts [76]. The

data provided by WINE is very valuable for the research community, because these hosts are computers that are actively used by real users which are potential victims of various cyber threats. WINE adopts a 1:16 sampling on this large-scale data such that all types of complex experiments can be held at scale.

To filter out from our analysis the binaries that are not good candidates to belong to malware development, we used two WINE datasets: the binary reputation and the AntiVirus telemetry datasets. The binary reputation dataset contains information about all of the executables (both malicious and benign) downloaded by Symantec customers over a period of approximately 5 years. To preserve the user privacy, this data is collected only from the users that gave explicit consent for it. At the time we performed our study, the binary reputation dataset included reports for over 400 millions of distinct binaries. On the other hand, the AntiVirus telemetry dataset records only the detections of known files that triggered the Norton Antivirus Engine on the users' machines.

The use of binary reputation helps us locating the exact point in time in which a binary was first disseminated in the wild. The AntiVirus telemetry data provided instead the first time the security company deployed a signature to detect the malware. We combined these datasets to remove those files that had already been observed by Symantec either before the submission to Anubis Sandbox, or within 24 hours from the time they were first submitted to the sandbox.

VirusTotal Filter

VirusTotal is a public service that provides virus scan results and additional information about hundreds of millions of analyzed files. In particular, it incorporates the detection results of over 50 different AntiVirus engines – thus providing a reliable estimation of whether a file is benign or malicious. Please note that we fetched the VirusTotal results for each file in our dataset several months (and in some cases even years) after the file was first submitted. This ensures that the AV signatures were up to date, and files were not misclassified just because they belonged to a new or emerging malware family.

Among all the information VirusTotal provides about binaries, the most important piece of information we incorporate in our study is the first submission time of a certain file to the service. We believe that by combining the timestamps obtained from the VirusTotal and Symantec datasets, we achieved an acceptable approximation of the first time a certain malicious file was observed in the wild.

3.4 Sample Analysis

If a sample survived the data reduction phase, it means that (with a certain approximation due to the coverage of Symantec and Virustotal datasets) it had never been observed in the wild before it was submitted to the online malware analysis sandbox. Although this might be a good indicator, it is still not sufficient

to flag the submission as part of a potential malware development. In fact, there could be other possible explanations for this phenomenon, such as the fact that the binary was just a new metamorphic variation of an already known malware family.

Therefore, to reduce the risk of mis-classification, in this paper we consider a candidate for possible development only when we can observe at least two samples that clearly show the changes introduced by the author in the software. In the rest of this section we describe how we find these groups of samples by clustering similar submissions together based on the sample similarity.

3.4.1 Sample Clustering

In the last decade, the problem of malware clustering has been widely studied and various solutions have been proposed [92, 97, 100, 201]. Existing approaches typically use behavioral features to group together samples that likely belong to the same family, even when the binaries are quite different. Our work does not aim at proposing a new clustering method for malware. In fact, our goal is quite different and requires to group files together only when they are very similar (we are looking for small changes between two versions of the same sample) and not when they just belong to the same family. Therefore, we leverage a clustering algorithm that simply groups samples together based on their binary similarity (as computed by *ssdeep* [114]) and on a set of features we extract from the submission metadata.

Moreover, we decided to put together similar binaries into the same cluster only if they were submitted to our sandbox in a well defined time window. Again, the assumption is that when a malware author is working on a new program, the different samples would be submitted to the online sandbox in a short timeframe. Therefore, to cluster similar binaries we compute the binary similarities among all the samples submitted in a sliding window of seven days. We then shift the sliding window ahead of one day and repeat this step. We employ this sliding window approach in order (1) to limit the complexity of the computation and the total number of binary comparisons, and (2) to ensure that only the binaries that are similar and have been submitted within one week from each other are clustered together. We also experimented with other window sizes (between 2 and 15 days) but while we noticed a significant reduction of clusters for shorter thresholds, we did not observed any advantage in increasing it over one week.

Similarities among binaries are computed using the *ssdeep* [114] tool which is designed to detect similarities on binary data. *ssdeep* provides a light-weight solution for comparing a large-number of files by relying solely on similarity digests that can be easily stored in a database. As we already discarded packed binaries in the data reduction phase, we are confident that the similarity score computed by *ssdeep* is a very reliable way to group together binaries that share similar code snippets. After computing the similarity metrics, we executed a simple agglomerative clustering algorithm to group the binaries for which the similarity score is greater than 70%. Note that this step is executed separately for each time window, but it

preserves transitivity between binaries in different sliding windows. For example, if file *A* is similar to *B* inside *window1*, and *B* is similar to file *C* inside the next sliding window, at the end of the process *A*, *B* and *C* will be grouped into the same cluster. As a result, a single cluster can model a malware development spanning also several months.

Starting from the initial number of binaries, we identified 5972 clusters containing an average of 4.5 elements each.

Inter-Cluster Relationships

The *ssdeep* algorithm summarizes the similarity using an index between 0 (completely different) and 100 (perfect match). Our clustering algorithm groups together samples for which the difference between the fuzzy hashes is greater than the 70% threshold. This threshold was chosen according to previous experiments [114], which concluded that 70% similarity is enough to guarantee a probability of misclassification close to zero.

However, if the malware author makes very large changes on a new version of his program, our approach may not be able to find the association between the two versions. Moreover, the final version of a malware development could be compiled with different options, making a byte-level similarity too imprecise. To mitigate these side effects, after the initial clustering step, we perform a refinement on its output by adding inter-clusters edges whenever two samples in the same time window share the same submission origin (i.e., either from the same IP address or using the same email address for the registration). These are "weak" connections that do not model a real similarity between samples, and therefore they are more prone to false positives. As a consequence, our system does not use them when performing its automated analysis to report suspicious clusters. However, as explained in Section 3.6, these extra connections can be very useful during the analysis of a suspicious cluster to gain a more complete picture of a malware development.

After executing this refinement step, we were able to link to our clusters an additional 10,811 previously isolated binaries. This procedure also connected several clusters together, to form 225 macro groups of clusters.

3.4.2 Intra-cluster Analysis

Once our system had clustered the binaries that likely belong to the same malware development, we investigate each cluster to extract more information about its characteristics. In particular, we perform a number of code-based analysis routines to understand if the samples in the same cluster share similar code-based features.

Code Normalization

Code normalization is a technique that is widely used to transform binary code to a canonical form [51]. In our study, we normalize the assembly code such that

the differences between two binaries can be determined more accurately. Under the assumption that two consecutive variations of the same program are likely compiled with the same tool chain and the same options, code normalization can be very useful to remove the noise introduced by small variations between two binaries.

There are several approaches that have been proposed to normalize assembly code [101, 109, 172]. Some of them normalize just the operands, some the mnemonics, and some normalize both. In this paper, we chose to normalize only the operands so that we can preserve the semantics of the instructions. In particular, we implemented a set of IDA Pro plugins to identify all the functions in the code and then replace, for each instruction, each operand with a corresponding placeholder tag: `reg` for registers, `mem` for memory locations , `val` for constant values, `near` for near call offsets, and `ref` for references to memory locations. These IDA scripts were run in batch mode to pre-process all the samples in our clusters.

Programming Languages

The second step in our intra-cluster analysis phase consists in trying to identify the programming language used to develop the samples. The programming language can provide some hints about the type of development. For example, scripting languages are often used to develop tools or probes designed to exfiltrate information from the sandbox. Moreover, it is likely that a malware author would use the same programming language for all the intermediate versions of the same malware. Therefore, if a cluster includes samples of a malware development, all samples should typically share the same programming language. Exceptions, as the one explained in Section 3.6, may point to interesting cases.

To detect the programming language of a binary we implemented a simple set of heuristics that incorporate the information extracted by three tools: `PEiD`, the `pefile` python library, and the Linux `strings` command. First, we use `pefile` to parse the Import Address Table (IAT) and obtain the list of libraries that are linked to the binary. Then, we search for programming language specific keywords on the extracted list. For example, the "VB" keyword in the library name is a good indicator of using Visual Basic, and including `mscoree.dll` in the code can be linked to the usage of Microsoft `.NET`. In the second step of our analysis, we analyze the strings and the output of `PEiD` to detect compiler specific keywords (e.g., `type_info` and RTTI produced by C++ compilers, or "Delphi" strings generated by the homonymous language).

With these simple heuristics, we identified the programming language of 14,022 samples. The most represented languages are Visual Basic (49%), C (21%), Delphi (18%), Visual Basic .Net (7%), and C++ (3%). The large number of Visual Basic binaries could be a consequence of the fact that a large number of available tools that automatically create generic malware programs adopt this language.

Fine-grained Sample Similarity

In this last phase, we look in more detail at the similarity among the samples in the same cluster. In particular, we are interested to know why two binaries show a certain similarity: Did the author add a new function to the code? Did she modify a branch condition, or remove a basic block? Or maybe the code is exactly the same, and the difference is limited to some data items (such as a domain name, or a file path).

To answer these questions, we first extract the timeline of each cluster, i.e., the sequence in which each sample was submitted to the sandbox in chronological order. Moving along the timeline, we compare each couple of samples using a number of static analysis plugins we developed for IDA Pro.

The analysis starts by computing and comparing the *call graph* of the two samples. In this phase we compare the normalized code of each function, to check which functions of the second binary were present unchanged in the first binary. The output is a list of additional function that were not present in the original file, plus a list of functions that were likely modified by the author – i.e., those function that share the same position in the call graph but whose code does not perfectly match. However, at this level of granularity it is hard to say if something was modified in the function or if the author just removed the function and added another with the same callee.

Therefore, in these cases, we "zoom" into the function and repeat our analysis, this time comparing their *control flow graphs* (CFGs). Using a similar graph-based approach, this time we look for differences at the basic block level. If the two CFGs are too different, we conclude that the two functions are not one the evolution of the other. Otherwise, we automatically locate the different basic blocks and we generate a similarity measure that summarize the percentage of basic blocks that are shared by the two functions.

3.4.3 Feature Extraction

Based on the analysis described in the previous sections, our system automatically extracts a set of 48 attributes that we believe are relevant to study the dynamics of malware development.

This was done in two phases. First, we enriched each sample with 25 individual features, divided in six categories (see the Appendix for a complete list of individual features). The first class includes self-explanatory file features (such as its name and size). The Timestamps features identify when the sample was likely created, when it was submitted to Anubis Sandbox, and when it was later observed in the wild. While the creation time of the binary (extracted from the PE headers) could be manually faked by the author, we observed that this is seldom the case in practice, in particular when the author submits a probe or an intermediate version of a program. In fact, in these cases we often observed samples in which the compilation time precedes the submission time by only few minutes.

The third category of features contain the output of the VirusTotal analysis on the sample, including the set of labels associated by all AntiVirus software and the number of AVs that flag the sample as malicious. We then collect a number of features related to the user who submitted the sample. Since the samples are submitted using a web browser, we were able to extract information regarding the browser name and version, the language accepted by the system (sometime useful to identify the nationality of the user) and the IP from which the client was connecting from. Two features in this set require more explanation. The email address is an optional field that can be specified when submitting a sample to the sandbox web interface. The proxy flag is instead an attempt to identify if the submitter is using an anonymization service. We created a list of IP addresses related to these services and we flagged the submissions in which the IP address of the submitter appears in the blacklist. In the Binary features set we record the output of the fine-grained binary analysis scripts, including the number of sections and functions, the function coverage, and the metadata extracted by the PE files. Finally, in the last feature category we summarize the results of the sandbox behavioral report, such as the execution time, potential runtime errors, use of evasion techniques, and a number of boolean flags that represent which behavior was observed at runtime (e.g., HTTP traffic, TCP scans, etc.)

In the second phase of our analysis we extended the previous features from a single sample to the cluster that contains it. Table 3.2 shows the final list of aggregated attributes, most of which are obvious extensions of the values of each sample in the cluster. Some deserve instead a better explanation. For instance, the cluster `shape` (A3) describes how the samples are connected in the cluster: in a tightly connected group, in a chain in which each node is only similar to the next one, or in a mixed shape including a core group and a small tail. The `Functions diff` (B13) summarized how many functions have been modified in average between one sample and the next one. `Dev time` (B25) tells us how far apart in time each samples were submitted to the sandbox, and `Connect Back` (B24) counts how many samples in the cluster open a TCP connection toward the same /24 subnetwork from which the sample was submitted. This is a very common behavior for probes, as well as for testing the data exfiltration component of a malicious program.

Finally, some features such as the number of crashes (C8) and the average VT detection (D4) are not very interesting per se, but they become more relevant when compared with the number of samples in the cluster. For example, imagine a cluster containing three very similar files. Two of them run without errors, while the third one crashes. Or two of them are not detected by AV signatures, but one is flagged as malware by most of the existing antivirus software.

While we are aware of the fact that each feature could be easily evaded by a motivated attacker, as described in Section 3.6 the combinations of all them is usually sufficient to identify a large number of development clusters. Again, our goal is to show the feasibility of this approach and draw attention to a new problem, and not to propose its definitive solution.

A: Cluster Features	
A.1 Cluster_id	The ID of the cluster
A.2 Num Elements	The number of samples in the cluster
A.3 Shape	An approximation of the cluster shape (GROUP|MIX|CHAIN)
B: Samples Features	
B.1-4 Filesize stats	Min, Max, Avg, and Variance of the samples filesize
B.5-8 Sections stats	Min, Max, Avg, and Variance of the number of sections
B.9-12 Functions stats	Min, Max, Avg, and Variance of the number of functions
B.13 Functions diff	Average number of different functions
B.14 Sections diff	Average number of different sections
B.15 Changes location	One of: Data, Code, Both, None
B.16 Prog Languages	List of programming languages used during the development
B.17 Filename Edit Distance	The Average edit distance of the samples's filenames
B.18 Avg Text Coverage	Avg text coverage of the .text sections
B.19-22 CTS Time	Min, Max, Avg, and Variance of the difference between compile and the submission time
B.23 Compile time Flags	Booleans to flag `NULL` or constant compile times
B.24 Connect back	True if any file in the cluster contacts back the submitter's /24 network
B.25 Dev time	Average time between each submission (in seconds)
C: Sandbox Features	
C.1 Sandbox Only	Numer of samples seen only by the sandbox (and not from external sources)
C.2 Short Exec	Number of samples terminating the analysis in less than 60s
C.4-6 Exec Time	Min, Max, and Avg execution time of the samples within the sandbox
C.7 Net Activity	The number of samples with network activity
C.7 Time Window	Time difference between first and last sample in the cluster (in days)
C.8 Num Crashes	Number of samples crashing during their execution inside the sandbox
D: Antivirus Features	
D.1-3 Malicious Events	Min, Max, Avg numbers of behavioral flags exibited by the samples
D.4-5 VT detection	Average and Variance of VirusTotal detection of the samples in the cluster
D.6 VT Confidence	Confidence of the VirusTotal score
D.7 Min VT detection	The score for the sample with the minimum VirusTotal Detection
D.8 Max VT detection	The score for the sample with the maximum VirusTotal Detection
D.9 AV Labels	All the AV labels for the identified pieces of malware in the cluster
E: Submitter Features	
E.1 Num IPs	Number of unique IP addresses used by the submitter
E.2 Num E-Mails	Number of e-mail addresses used by the submitter
E.3 Accept Languages	Accepted Languages from the submitter's browser

Table 3.2 – List of Features associated to each cluster

3.5 Machine Learning

Machine learning provides a very powerful set of techniques to conduct automated data analysis. As the goal of this paper is to automatically distinguishing malware developments from other submissions, we tested with a number of machine learning techniques applied to the set of features we presented in detail in the previous section.

	AUC	Det. Rate	False Pos.
Full data	0.999	98.7%	0%
10-folds Cross-Validation	0.988	97.4%	3.7%
70% Percentage Split	0.998	100%	11.1%

Table 3.3 – Classification accuracy, including detection and false positive rates, and the Area Under the ROC Curve (AUC)

Among the large number of machine learning algorithms we have tested our training data with, we have obtained the best results by using the logistic model tree (LMT). LMT combines the logistic regression and decision tree classifiers by building a decision tree whose leaves have linear regression models [121].

Training Set

The most essential phase of machine learning is the training phase where the algorithm learns how to distinguish the characteristics of different classes. The success of the training phase strictly depends on a carefully prepared labeled data. If the labeled data is not prepared carefully, the outcome of machine learning can be misleading. To avoid this problem, we manually labeled a number of clusters that were randomly chosen between the ones created at the end of our analysis phase. Manual labeling was carried out by an expert that performed a manual static analysis of the binaries to identify the type and objective of each modification. With this manual effort, we flagged 91 clusters as non-development and 66 as development. To estimate the accuracy of the LMT classifier, we conducted a 10-fold cross validation and a 70% percentage split evaluation on the training data.

Feature Selection

In the previous section, we have presented a comprehensive set of features that we believe can be related to the evolution of samples and to distinguish malware developments from ordinary malware samples. However, not all the features contribute in the same way to the final classification, and some works well only when used in combination with other classes.

To find the subset of features that achieves the optimal classification accuracy while helping us to obtain the list of features that contribute the most to it, we leveraged a number of features selection algorithms that are widely used in machine learning literature: Chi-Square, Gain Ratio and Relief-F attribute evaluation. Chi-square attribute evaluation computes the chi-square statistics of each feature with respect to the class, which in our case is the fact of being a malware development or not. The Gain Ratio evaluation, on the other hand, evaluates the effect of the feature by measuring its gain ratio. Finally, the Relief-F attribute evaluation methodology assigns particular weights to each feature according to how much

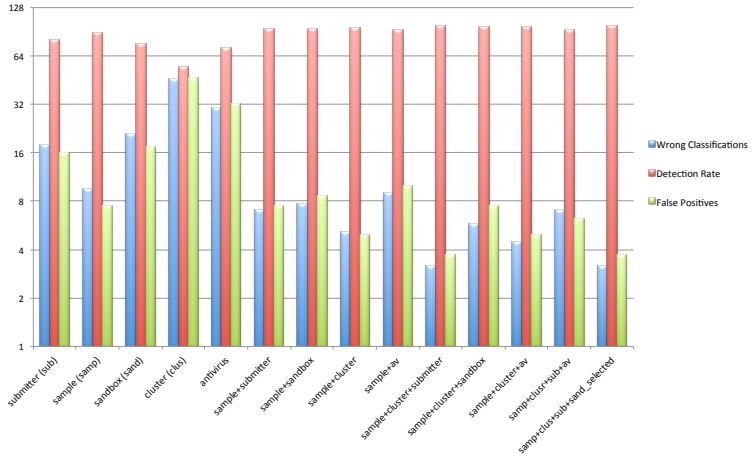

Figure 3.1 – Classification success of different feature combinations.

they are successful to distinguish the classes from each other. This weight computation is based on the comparison of the probabilities of two nearest neighbors having the same class and the same feature value.

While the order slightly differs, the ten most effective features for the accuracy of the classifier for all three feature selection algorithms are the same. As also the common sense suggests, the features we extract from the binary similarity and the analysis of the samples are the most successful. For example, the connect back feature that checks if the sample connects back to the same IP address of the submitter, the average edit distance of the filenames of the samples, the binary function similarity, and the sample compile time features are constantly ranked on the top of the list. The submitter features and the sandbox features are following the sample features in the list. All of the features except the number of sandbox evasions, the VirusTotal results, and the features we extracted from the differences on the file sizes in the clusters had a contribution to the accuracy. After removing those features, we performed a number of experiments on the training set to visualize the contribution of the different feature sub-sets to the classification accuracy. Figure 3.1 shows (in log-scale) the impact of each class and combination of classes. Among all the classes the samples-based features produced the best combination of detection and false positive rates (i.e. 88.2% detection rate with 7.4% false positives). In particular, the ones based on the static and dynamic analysis of the binaries seem to be the core of the detection ability of the system. Interestingly, the cluster-based features alone are the worst between all sets, but they increase the accuracy of the final results when combined with other features.

The results of the final classifier are reported in Table 3.3: 97.4% detection with of 3.7% false positives, according to 10-folds cross validation experiment.

Campaign	Early Submission	Time Before Public Disclosure	Submitted by
Operation Aurora	✓	4 months	US
Red October	✓	8 months	Romania
APT1	✓	43 months	US
Stuxnet	✓	1 months	US
Beebus	✓	22 months	Germany
LuckyCat	✓	3 months	US
BrutePOS	✓	5 months	France
NetTraveller	✓	14 months	US
Pacific PlugX	✓	12 months	US
Pitty Tiger	✓	42 months	US
Regin	✓	44 months	UK
Equation	✓	23 months	US

Table 3.4 – Popular campaigns of targeted attacks in the sandbox database

Note that we decided to tune the classifier to favor detection over false positives, since the goal of our system is only to tag suspicious submissions that would still need to be manually verified by a malware analyst.

3.6 Results

Our prototype implementation was able to collect substantial evidences related to a large number of malware developments.

In total, our system flagged as potential development 3038 clusters over a six years period. While this number was too large for us to perform a manual verification of each case, if such a system would be deployed we estimate between two and three alerts generated per day. Therefore, we believe our tool could be used as part of an early warning mechanism to automatically collect information about suspicious submissions and report them to human experts for further investigation.

In addition to the 157 clusters already manually labeled to prepare the training set for the machine learning component, we also manually verified 20 random clusters automatically flagged as suspicious by our system. Although according to the 10-fold cross validation experiments the false positive rate is 3.7%, we have not found any false positives on the clusters we randomly selected for our manual validation.

Our system automatically detected the development of a diversified group of real-world malware, ranging from generic trojans to advanced rootkits. To better understand the distribution of the different malware families, we verified the AV labels assigned to each reported cluster. According to them, 1474 clusters were classified as malicious, out of which our system detected the development of 45 botnets, 1082 trojans, 83 backdoors, 4 keyloggers, 65 worms, and 21 malware de-

velopment tools (note that each development contained several different samples modeling intermediate steps). A large fraction of the clusters that were not identified by the AV signatures contained the development of probes, i.e., small programs whose goal is only to collect and transmit information about the system where they run. Finally, some clusters also contained the development or testing of offensive tools, such as packers and binders.

3.6.1 Targeted Attacks Campaigns

Before looking at some of the malware development cases detected by our system, we wanted to verify our initial hypothesis that even very sophisticated malware used in targeted attacks are often submitted to public sandboxes months before the real attacks are discovered. For this reason, we created a list of hashes of known and famous APT campaigns, such as the ones used in operation Aurora and Red October. In total, we collected 1271 MD5s belonging to twelve different campaigns. As summarized in Table 3.4, in all cases we found at least one sample in our database before the campaign was publicly discovered (*Early Submission* column). For example, for Red October the first sample was submitted in February 2012, while the campaign was later detected in October 2012. The sample of Regin was collected a record 44 months before the public discovery.

Finally, we checked from whom those samples were submitted to the system. Interestingly, several samples were first submitted by large US universities. A possible explanation is that those samples were automatically collected as part of a network-based monitoring infrastructure maintained by security researchers. Other were instead first submitted by individual users (for whom we do not have much information) from several different countries, including US, France, Germany, UK, and Romania. Even more interesting, some were first submitted from DSL home Internet connections. However, we cannot claim that we observed the development phase of these large and popular targeted attacks campaigns as in all cases the samples were already observed in the wild (even though undetected and no one was publicly aware of their existence) before they were submitted to our sandbox. It is important to note that for this experiment we considered the entire dataset, without applying any filtering and clustering strategy. In fact, in this case we did not want to spot the *development* of the APT samples, but simply the fact that those samples were submitted and available to researchers long before they were publicly discovered.

We believe the sad message to take away from this experiment is that all those samples went unnoticed. As a community, there is a need for some kind of early warning system to report suspicious samples to security researches. This could prevent these threats from flying under the radar and could save months (or even years) of damage to the companies targeted by these attacks.

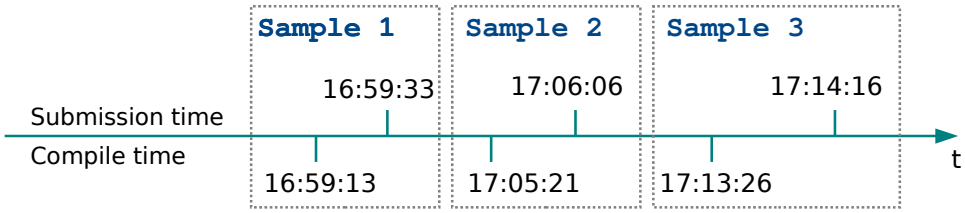

Figure 3.2 – Anti-sandbox check - Timeline

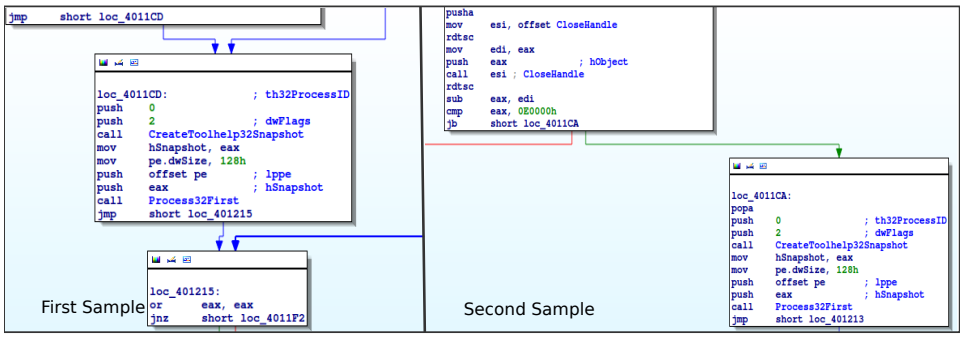

Figure 3.3 – Anti-sandbox check - Start function comparison

3.6.2 Case studies

In the rest of this section we describe in more details three development scenarios. While our system identified many more interesting cases, due to space limitation we believe the following brief overview provides a valuable insight on the different ways in which attackers use (and misuse) public sandboxes. Moreover, it also shows how a security analyst can use the information collected by our system to investigate each case, and reconstruct both the author behavior and his final goal.

In the first example, the malware author introduced an anti-sandbox functionality to a Trojan application. In this case the analyst gathers intelligence information about the modus operandi of the attacker and about all the development phases.

In the second scenario, we describe a step by step development in which the attacker tries to collect information from the sandbox. This information is later used to detect the environment and prevent the execution of a future malware in the sandbox. In the last example, we show how an attacker uses the sandbox as a testbed to verify the behavior of the malware. In this case, the author generated the binary using one of the many dedicated builder applications that can be downloaded from the Internet or bought on the black market.

Example I: Anti-sandbox Malware

The cluster related to this example contains three samples. The timeline (summarized in Figure 3.2) already suggests a possible development. In fact, the difference between the `submission` time and the `compile` time is very small.

A quick look at the static features of the cluster shows that the three samples are very similar, and share the same strings as well as the same imphash (the *import hash* [29, 30] recently introduced also by VirusTotal). However, the first sample is composed of 21 functions, while the last two samples have 22 functions. Our report also shows how the first and the second samples differ for two functions: the author modified the function `start`, and introduced a new function `CloseHandle`. This information (so far extracted completely automatically by our system) is a good starting point for a closer analysis.

We opened the two executables in IDA Pro, and quickly identified the two aforementioned functions (snippet in Figure 3.3). It was immediately clear that the `start` function was modified to add an additional basic block and a call to the new `CloseHandle` function. The new basic block uses the `rdtsc` x86 instruction to read the value of the Timestamp Counter Register (TSC), which contains the number of CPU cycles since the last reset. The same snippet of assembly is called two times to check the time difference. After the first `rdtsc` instruction there is a call to `CloseHandle`, using the timestamp as handler (probably an invalid handler). These two well known tricks are here combined to detect the Anubis Sandbox environment – due to the delay introduced by its checks during program execution. The Anubis Sandbox's core is slower in looking up the handlers table, and this time discrepancy is the key to detect the analysis environment. In this case the difference has to be less than `0E0000h`, or the program would immediately terminate by calling the `ExitProcess` function.

The last sample in the cluster was submitted only to tune the threshold and for this reason there were no important differences with the second sample. The *control flow graph* analysis performed automatically by our system report a very high similarity between the first two samples, in line with the little modifications we found in the disassembled code. Finally, the behavioral features extracted by our system confirm our hypothesis: the first sample was executed until the analysis timeout, but the execution of the second one terminated after only five seconds.

The behavior described so far suggest malicious intents. This is also confirmed by other cluster metadata. For instance, while the first sample in the cluster was unknown to VirusTotal, the last one was clearly identified as a common `Trojan` application. This suggests that the original sample, without the timing check, has never been used in the wild. Once more, the fact that all three samples have been submitted days before the trojan was first observed in the wild strongly supports the fact that the person who submitted them was indeed the malware author.

Example II: Testing a Trojan Dropper

The second cluster we want to describe is composed of five samples. Our report indicates that the first four are written in Delphi and the last one is written in Visual Basic. This is already a strange fact, since the two programming languages are quite different and it is unlikely that they could generate similar binaries.

In this case the cluster timeline does not provide useful information as all the Delphi samples share exactly the same compilation time: 20th of June, 1992. Only

the Visual Basic sample had a compilation time consistent with the submission. On the contrary, the submission times provide an interesting perspective. All the samples have been submitted in few hours and this might indicate a possible development. In addition, there are two IP addresses involved: one for the four Delphi samples and one for the final Visual Basic version. The static features of the first four samples show very little differences, suggesting that these are likely just small variations of the same program. In average, they share 169 out of 172 functions and 7 out of 8 PE sections. By inspecting the changes, we notice that the attacker was adding some threads synchronization code to a function responsible for injecting code into a different process. The *control flow graph* similarity reported by our tool was over 98%, confirming the small differences we observed between each versions. Once the author was happy with the result, she submitted one more sample, this time completely different from the previous ones. Despite the obvious differences in most of the static analysis features, the fuzzyhash similarity with sample 4 was 100%. A rapid analysis showed that this perfect match was due to the fact that the Visual Basic application literally embedded the entire binary of the fourth Delphi program. In addition, the behavior report confirmed that, once executed, the Visual Basic Trojan dropped the embedded executable that was later injected inside a target process. None of the Antivirus software used by VirusTotal recognized the first four samples as malicious. However, the last one was flagged by 37 out of 50 AVs as a *trojan dropper* malware.

It is important to stress that a clear advantage of our system is that it was able to automatically reconstruct the entire picture despite the fact that not all samples were submitted from the same IP address (even though all located in the same geographical area). Moreover, we were able to propagate certain metadata extracted by our system (for example the username of the author extracted from the binary compiled with Visual Studio) from one sample to the others in which that information was missing. This ability to retrieve and propagate metadata between different samples can be very useful during an investigation.

Another very interesting aspect of this malware development is the fact that after the process injection, the program used a well known dynamic DNS service (`no-ip`) to resolve a domain name. The IP address returned by the DNS query pointed exactly to the same machine that was used by the author to submit the sample. This suggests that the attacker was indeed testing his attack before releasing it, and this information could be used to locate the attacker machine.

We identified a similar *connect-back* behavior in other 1817 clusters. We also noticed how most of these clusters contain samples generated by known trojan *builders*, like Bifrost [16] or PoisonIvy [18]. While this may seem to prove that these are mostly unsophisticated attacks, FireEye [31] recently observed how the Xtremerat builder [14] (which appeared in 28 of our clusters) was used to prepare samples used in several targeted attacks.

Example III: Probe Development

In this last example we show an attacker fingerprinting the analysis environment and how, at the end, she manages to create her own successful antisandbox check. The cluster consists of two samples, both submitted from France in a time span of 23 hours by the same IP address. The two samples have the same size, the same number of functions (164), and of sections (4). There is only one function (`_start`) and two sections (`.text` and `.rdata`) presenting some differences. The two programs perform the same actions, they create an empty text file and then they retrieve the file attributes through the API `GetFileAttributes`. The only differences are on the API version they use (`GetFileAttributesA` or `GetFileAttributesW`) and on the file name to open.

At a first look, this cluster did not seem very interesting. However the inter-cluster connections pointed to other six loosely correlated samples submitted by the same author in the same week. As explained in Section 3.4, these files have not been included in the core cluster because the binary similarity was below our threshold. In this case, these samples were all designed either to collect information or to test anti-virtualization/emulation tricks. For instance, one binary implemented all the known techniques based on `idt`, `gdt` and `ldt` to detect a virtual machine monitor [131, 166, 169]. Another one simply retrieved the computer name, and another one was designed to detect the presence of inline hooking. Putting all the pieces together, it is clear that the author was preparing a number of probes to assess various aspects of the sandbox environment.

This example shows how valuable the inter-clusters edges can be to better understand and link together different submissions that, while different between each other at a binary level, are likely part of the same organized "campaign".

3.6.3 Malware Samples in the Wild

As we already mentioned at the beginning of the section, out of 3038 clusters reported as malware development candidates by our machine learning classifier, 1474 (48%) contained binaries that were detected by the antivirus signatures as malicious (according to VirusTotal).

A total of 228 of the files contained in these clusters were later detected in the wild by the Symantec's antivirus engine. The average time between the submission to our sandbox and the time the malware was observed in the wild was 135 days – i.e., it took between four and five months for the antivirus company to develop a signature and for the file to appear on the end-users machines. Interestingly, some of these binaries were later detected on more than 1000 different computers in 13 different countries all around the world (obviously a lower bound, based on the alerts triggered on a subset of the Symantec's customers). This proves that, while these may not be very sophisticated malware, they certainly have a negative impact on thousands of normal users.

3.7 Limitations

We are aware of the fact that once this research is published, malware authors can react and take countermeasures to sidestep this type of analysis systems. For instance, they may decide to use "private" malware checkers, and avoid interacting with public sandboxes altogether. First of all, this is a problem that applies to many analysis techniques ranging from botnet detection, to intrusion prevention, to malware analysis. Despite that, we believe that it is important to describe our findings so that other researchers can work in this area and propose more robust methodologies in the future.

Moreover, as we mentioned in the introduction, after we completed our study someone noticed that some known malware development groups were testing their creation on VirusTotal [71, 206]. This confirms that what we have found is not an isolated case but a widespread phenomenon that also affects other online analysis systems. Second, now that the interaction between malware developers and public sandboxes is not a secret anymore, there is no reason that prevents us from publishing our findings as well.

We are aware of the fact that our methodology is not perfect, that it can be evaded, and that cannot catch all development cases. However, we believe the key message of the paper is that malware authors are abusing public sandboxes to test their code, and at the moment we do not need a very sophisticated analysis to find them. Since this is the first paper that tries to identify these cases, we found that our approach was already sufficient to detect thousands of them. Certainly more research is needed in this area to develop more precise monitoring and early warning system to analyze the large amounts of data automatically collected by public services on a daily basis.

Chapter 4

Network Containment in Malware Analysis Systems

4.1 Introduction

The approach proposed in this chapter addresses the repeatability and the containment of malware execution by exploring the use of protocol learning techniques for the emulation of the external network environment required by malware samples.

In particular, we address two problems. The first is the poor repeatability of malware analysis experiments. Malware analysts often execute samples inside a sandbox, in order to observe and collect their malicious behavior. However, the exhibited behavior may depend on external factors, such as the commands received by a C&C server or the content of a given URL. For example, consider a classic scenario common to many security companies. Collected samples are automatically analyzed by a malware analysis system, and their behavior (e.g., filesystem operations, process creations, and modification to the Windows registry) is stored in a database. When a program requires a closer look, an analyst can run it again in a separate, better instrumented environment, for example by using a debugger or by collecting all the system calls. Unfortunately, it is often the case that these "secondary inspections" are performed several days, or even weeks after the samples were initially collected, with the risk of studying dead samples for which the required infrastructure is not available anymore. In fact, the remote machines contacted by malware are volatile by nature, often hosted on other compromised computers, or taken down by providers and law enforcement when the malicious activity is detected. Therefore, the malware infrastructure required to properly run the sample is normally available for only a limited amount of time.

The second problem we address in this paper is related to malware containment, i.e., to the ability of properly execute a given sample in an isolated environment, where it cannot cause any harm to the rest of the world. In general, full containment of a new, previously unknown, sample is impossible. However, we

can identify two scenarios in which such result could be achieved: the execution of a polymorphic variation of an already analyzed malware, and the re-execution of a previously studied sample. In both cases, the sample (or a behaviorally equivalent variation of it) has already been analyzed by the system and, therefore, the problem of full containment can be reduced to the previous problem of repeatable execution. The idea is that if we can "mimic" the behavior of the network to match the one observed in a previous execution, we can obtain at the same time a repeatable experiment and a containment of the malware execution.

To mitigate these issues, in this chapter we want to study to which extent it is possible to enrich the information collected during malware execution to make the experiments repeatable and achieve full network containment. In particular, we experiment with a protocol-agnostic technique [128] previously adopted to model the attack traffic in high-interaction honeypots [124, 126]. The idea consists in building a finite state machine (FSM) of the network activity generated by each malware sample. The extracted FSM can be stored alongside the other collected information, and can then be used to properly "simulate" all the required endpoints whenever the sample needs to be analyzed again in the future.

It is important to note that we only address *network* repeatability: Our goal is to ensure that the malware finds all the remote components it needs to properly execute. On the contrary, we do not address full process repeatability, i.e., the problem of forcing two executions of a malicious executable to behave exactly the same from an operating system point of view. Balzarotti et al. [38] studied the problem of full repeatability in the context of the detection of split personalities in malware. Their prototype works at the system-call level and has several technical limitations, making it hard to deploy on current malware sandboxes.

To summarize, we make the following contributions:

— We discuss how protocol learning techniques can be used to model the traffic generated during the execution of malware samples. In particular, we describe the limitations of protocol-agnostic approaches and show that, if properly setup and configured, they can be used to successfully replay real malware conversations.

— We describe the implementation of Mozzie, a network containment system that can be easily applied to all the existing sandbox environments. According to our experiments, an average of 14 network traces are required by Mozzie to model the traffic and achieve full containment for real malware samples.

4.2 Protocol Inference

A common problem when looking at the network interaction generated by a malware sample is associated to the interpretation of application-level protocols. Malware samples often propagate by exploiting vulnerabilities in poorly documented protocols (such as SMB), and rely on custom protocols for their coordi-

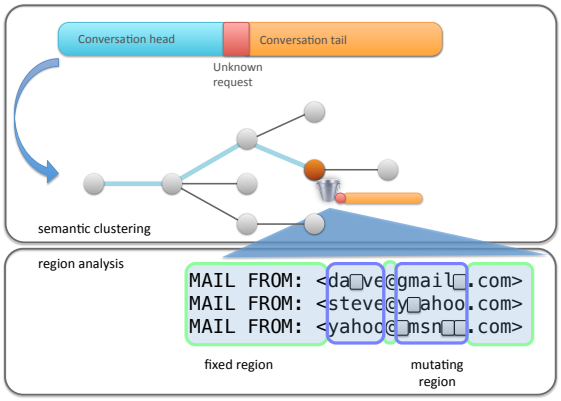

Figure 4.1 – Simplified diagram of the ScriptGen operation

nation with the C&C servers [190–192]. The execution of the sample in isolation requires us to trick the malware sample in interacting with replicas of the real Internet hosts the malware interacts with (victims, C&C servers, ...), but without being able to assume a-priori knowledge of the application-level protocol implemented in such services.

This challenge is addressed in this paper by resorting to protocol learning techniques. Techniques such as ScriptGen [127, 128], RolePlayer [65], Discoverer [63] and Netzob [21] aim at partially reconstructing the protocol message syntax and, in some cases, the protocol Finite State Automata [21, 127] from network interactions. The inference is performed by looking at network traces generated by clients and servers while minimizing the number of assumptions on the protocol characteristics. Differently from other approaches such as [53, 132, 134, 202], that factor into the protocol analysis also host-based information obtained through execution monitoring or memory tainting, the above methods focus on the extraction of the protocol format solely from network traces and are therefore particularly suitable to our task, where we are unable to control the remote endpoints contacted by malware.

While any of the previouly mentioned tools would be suitable for this task, in this paper we focus on ScriptGen [127, 128] because of its limited amount of assumptions on the protocol characteristics and its support to the inference not only of single protocol messages, but also of their structure within the protocol flow. ScriptGen is an algorithm that generates an approximation of a protocol language by means of a “server-side” Finite State Machine whose scope corresponds to a specific TCP connection or UDP flow: the FSM root corresponds to the establishment of the connection, while any of the FSM leaves corresponds to the point in which the connection is successfully closed. In the FSM representation each transition is labelled with a regular expression matching a possible client request, while each state is labelled with the server answer to be sent back to the client when reaching that specific state.

ScriptGen performs this task through two subsequent processes, as illustrated in Figure 4.1:

1. **Semantic clustering.** Initially introduced in [127], the semantic clustering algorithm aims at grouping together protocol messages likely to be semantically similar. Any new conversation to be added to an existing protocol model will consist of two parts: an initial part (*head*) composed of messages already modeled in the current version of the FSM, and a final part (*tail*) composed of messages that do not match the current model. A 0-length head represents a conversation whose first message already differs from the current protocol model, while a 0-length tail models a conversation that already fully matches the existing protocol knowledge. Semantic clustering aims at grouping together messages sharing the same head (by matching the head messages against the current FSM model) and likely to be sharing the same tail (by clustering conversations sharing the same head according to the length of the tail messages).
2. **Region analysis.** Semantic clustering leads to the identification of protocol messages that are very likely to be associated to a specific semantic context. The process that generalizes the specific protocol messages into a set of regular expressions able to correctly recognize future messages sharing the same semantic value is called region analysis and was introduced in [128]. Through the analysis of multiple versions of a semantically similar message, the region analysis algorithm aims at identifying *regions* with similar characteristics. Through the subsequent application of global alignment algorithms (Needleman-Wunsch [152]), ScriptGen refines the clustering of the messages separating those exposing major structural differences (macroclustering) while correctly dealing with variable length fields. The final result consists in a global alignment of each identified group. The statistical analysis of each byte of the alignment outcome is used to identify protocol *regions*, i.e., portions of protocol message likely to have a specific semantic value. The distribution of the content of each region over the different samples gives us information on their semantic nature: regions with random content throughout the set of samples are likely to be nonce values, regions with fixed content are likely to be separators or semantically reach protocol fields, and regions mutating through a limited amount of possible values are likely to be associated to more subtle protocol semantics, whose preservation is decided through a *microclustering* threshold. The final outcome of this semantic evaluation is a set of regular expressions, each of which will lead to the generation of a new subtree in the FSM object.

Most of the ScriptGen operation is driven by thresholds, that regulate for instance the different clustering steps. A simple approach for tuning the thresholds to the best configuration was introduced in [128] and consisted in brute-forcing all the possible combinations of thresholds to identify the global optimum. While this is an computationally expensive process, the computed thresholds proved to

be sufficiently robust to handle protocols with "similar" characteristics in terms of amount of variability in their structure.

It is important to understand that ScriptGen avoids any assumption on the nature of the protocol separators or on the possible representation of semantically relevant fields, and performs a partial reconstruction of the protocol semantics by analyzing at the same time multiple samples of the same type of protocol exchange (*conversation*). The higher the number of conversations available to the algorithm, the more precise the protocol inference process will be. Intuitively, if we consider two message instances of a protocol containing a random cookie value, their cookie value could be *aabd* and *awed*. ScriptGen would have no way to consider such protocol section as a mutating region of 4 characters, since (by accident) both values start with 'a' and end with 'd'. These accidental false inferences can be filtered out only by considering a sufficiently large number of conversation samples.

ScriptGen has been successfully used in the past to automatically generate models of network behavior for the emulation of vulnerable services in exploits. This emulation was included in a distributed honeypot [124, 126] deployment, in order to emulate 0-day exploits and collect information on the propagation vector of self-propagating malware. This paper investigates the use of ScriptGen in a different context, that of malware analysis, and tries to leverage its properties to automatically generate network models for the remote endpoints involved in the execution of a malware sample in a sandboxed environment.

4.3 System Overview

Our approach to achieve repeatability and containment in malware analysis experiments can be summarized in four steps:

1. **Traffic Collection** - In this phase the system collects a number of network traces associated with the execution of a certain malware sample. This can be done by running the malware in a sandbox, or by extracting existing traces generated by past analyses.
2. **Endpoint Analysis** - This is a cleaning and normalization process applied to all the collected traces. Its main goal consists in removing anomalous traces that could affect the results and the associated conclusions. Each trace is then normalized to remove endpoint randomization, such as the one introduced by IP fluxing techniques.
3. **Traffic Modeling** - This phase aims at the automated generation of models starting from the collected traffic samples and at their subsequent storage in a compact representation. The modeling can be performed in two different ways: in an online fashion (from now on called *incremental learning*), in which the model is initially very simple and it is subsequently refined at every new execution of the sample, or in an *offline* fashion, more suitable for the analysis of previously collected network dumps. The actual logic

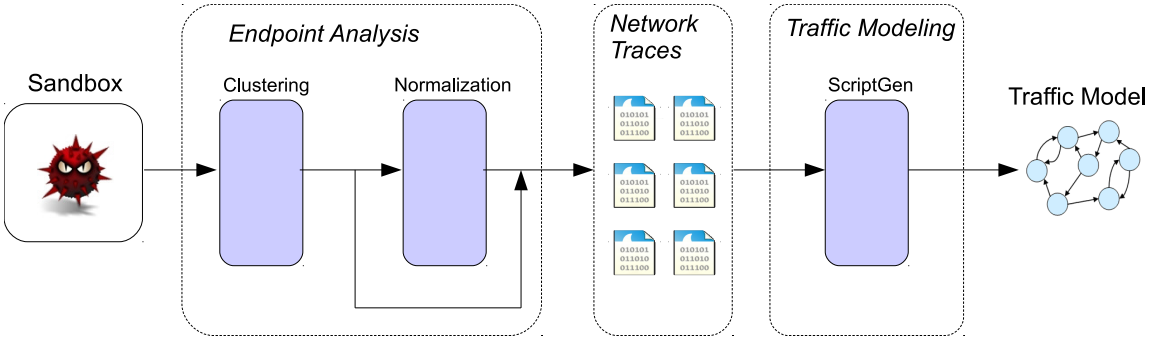

Figure 4.2 – Creation of a Traffic Model

used to model the traffic is implemented in a separate component of our system. As we already explained in Section 4.2, the current implementation is based on the ScriptGen approach, but other unsupervised algorithms can be easily plugged into our system to achieve the same result.

4. **Traffic Containment** - In this last phase, we use the model extracted in the previous step to mimic the network environment required by the malware sample. The containment system is implemented as a transparent proxy. When the model is sufficiently precise, the proxy is able to mimic the external world, effectively achieving "full containment". When the model is incomplete, the proxy redirects the requests it cannot handle to the real targets. In this case, the system also collects the forwarded traffic to improve the training set, effectively closing the loop back to Step 1.

In the rest of the section, we introduce each phase in details, and we describe how each of them have been implemented in our system.

4.3.1 Traffic Collection

Collecting the malware traffic is as simple as running a network sniffer while the sample is running in the sandbox. For instance, several online systems (e.g., Anubis [19], and CWSandbox [20]) allow users to download the pcap file recorded during the analysis phase.

As explained in Section 6.5, we used two different datasets for our experiments, one extracted from old Anubis reports, and one collected live by running the samples inside a Cuckoo's sandbox. Finally, in order to be consistent with the data collected in the past, also in our experiments we limited the malware analysis and the network collection time to five minutes per sample.

4.3.2 Endpoint Analysis

The second phase of our process consists in cleaning and normalizing the collected traffic to remove spurious traces and improve the effectiveness of the protocol learning phase when facing network-level randomization.

The cleaning phase mainly consists in grouping together traces that exhibit a comparable network behavior. The intuition underneath this cleaning process is that the traces may have been generated at different points in time, and may capture different "states" of the remote endpoints. Most of the traces are likely to have been generated when the malware was indeed fully active, but we may still have to deal with a minority of traces that may have been generated, for example, when the C&C server was temporarily not reachable. It should be noted that the semantic clustering process explained in Section 4.2 allows ScriptGen to correctly deal with these cases. However, in this paper we are interested in evaluating the efficiency of our method at correctly leveraging *useful* traces to generate usable models, and thus we choose to clean the dataset from these spurious traces. In practice, this is achieved by clustering together traces according to each involved destination endpoint, where endpoint is defined as an $(IP, port)$ tuple. We consider the cluster with the highest amount of traces having similar high-level network behavior as the one representing the state of interest for our experiments.

While the cleaning process succeeds in most of the cases, in some of our experiments we noted that the clustering algorithm failed to identify a predominant network behavior for a specific sample. Closer investigation revealed that this failure is associated to the introduction of randomization in the network behavior of the sample. The most common example of this phenomenon is associated to malware using IP fluxing techniques. IP flux, also known as fast-flux, is a DNS-based technology used by malware writers to improve the resilience of their (often botnet) architecture. The idea is to rapidly swap the IP address associated to a particular domain name, to avoid having a single-point of failure and reducing the effect of IP blacklisting.

When a malware uses IP fluxing, most of the collected network traces will involve different endpoints. In other words, instead of having ten samples of conversations associated to a single endpoint, we will have ten different targets associated to one conversatione each. As we will see in the discussion of the Traffic Modeling component, this situation plays against our choice of creating protocol models on a per-endpoint basis: each endpoint will not be associated to a sufficient amount of samples to generate a meaningful model. However, this phenomenon is easy to detect because our endpoint clustering component would return an unusual result composed of many clusters containing a single trace each. In this case we automatically "normalize" the endpoints by identifying the DNS request that returned different IP addresses and by forcing it to return always the same value. In these cases, our system automatically replaces each fluxed IP with the normalized IP in all the subsequent network flows. By performing this simple step we can obtain an uniform learning dataset, ready to be analyzed by the next stage of our system.

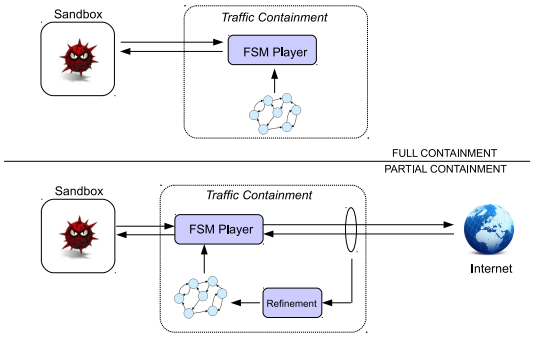

Figure 4.3 – Replaying Architecture

4.3.3 Traffic Modeling

Starting from the set of network traces obtained from the previous phase, the Traffic Modeling phase leverages the protocol learning algorithm (in our case, ScriptGen) to generate models for the network interaction with each endpoint involved in the malware execution. The models are maintained in the form of a dictionary, where each encoutered destination IP address and destination port is associated to its corresponding model (see Figure 4.2). This dictionary is later used to mimic the whole network environment, with the goal of containing all the requests generated by the malware.

4.3.4 Containment Phase

The goal of the containment phase is to "trick" the malware sample into believing that it is connected to the Internet, while in fact all the traffic is artificially generated by leveraging the information contained in the current protocol model.

The main component of the containment phase is the FSM player. The player is responsible for analyzing the incoming packets looking for new connections attempts, and checking if the contacted endpoint is present in the dictionary of FSMs. If so, the corresponding model is loaded and associated to the connection. Then, for each message, the player analyzes the current state and computes the next state. This is done by finding the best transition that matches the incoming content and by extracting the corresponding list of possible answers.

Figure 4.3 shows the operation of our system over two possible operation modes: *full* and *partial containment*.

In the *full containment* case, the protocol model is accurate enough to allow our system to correctly generate a response for every network interaction generated by the malware upon execution.

In the *partial containment* case, instead, the protocol model is inaccurate or incomplete: some of the network interactions created by the malware are not modelled in the associated FSM. Whenever a message cannot be matched with the current FSM, the system is unable to further emulate the associated remote end-

point. In such case, the system enters in replay mode for that specific endpoint, and replays all the traffic generated so far towards the real Internet host associated to it. By replaying all the traffic generated so far, we are able to handle possible authentication steps that the malware may have already performed in its interaction with the FSM. The answer to the unknown request is eventually delivered by the real endpoint, and all the subsequent interaction is then relayed by the system (proxy mode).

This process needs to be carried out without affecting the malware execution. For instance, consider the example in Figure 4.4 in which the malware sends three messages $\{Msg_1, Msg_2, Msg_3\}$ over a TCP connection. The player is able to follow the first two messages on the FSM, thus returning the corresponding responses $\{Resp_1, Resp_2\}$. However, the third message is different from what it was expecting, and it does not know what to answer. Therefore, Mozzie opens a new TCP connection to the original target, and quickly replay the messages 1 and 2 in chronological order to bring the new connection to the same state of the one it is simulating with the malware. Then it sends the Msg_3 and switches to proxy mode. In proxy mode, the system acts like a transparent proxy, forwarding each packet back and forth from the malware to the endpoint on the Internet. When the connection is terminated, the data is used to incrementally improve the model, so that it could handle the same conversation in the future.

By executing a malware sample multiple times, we are therefore able to gradually and automatically move from partial containment (in case in which part of the generated interacton is still unknown) to full containment, where all the malware network behavior is fully modeled and emulated by the system. However, it should be noted that the full learning of the network behavior of the malware is different from the full knowledge of the protocol. It is possible to follow a malware without contacting the external server in all its executions within the sandbox, but this does not mean all the commands of the protocols have been discovered.

4.3.5 System Implementation

In the previous sections we explained the architecture of the system. Now we can describe how Mozzie, our prototype implementation, is realized.

Mozzie is based on *iptables*. In particular it uses the NFQUEUE userspace packet handler with the Python nfqueue-bindings [17]. This allows our user-space component to accept, drop, or modify each incoming packet. To decode the packets that are in the queue, we used the Scapy [15] library.

Our current prototype handles three different IP protocols: ICMP, UDP and TCP. ScriptGen cannot model the ICMP protocol, because it lacks the concept of *port* that is required to build the Finite State Machine. Therefore, Mozzie intercepts all the ICMP ECHO request messages and always answers with an ECHO reply. Beside that, the system mainly works as a userspace NAT, changing the destination IP and port for each TCP or UDP packet, to redirect them to the emulator responsible for that endpoint. The emulator runs an implementation of the Script-

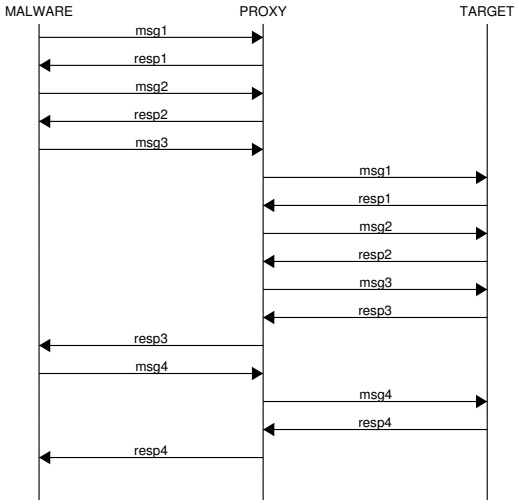

Figure 4.4 – Sequence of messages during traffic replay

Gen algorithm and one instance of the FSM Player for each endpoint contacted by the malware. For example, if the malware opens a new TCP connection toward $(IP, Port)$, Mozzie checks if a FSM exists for that endpoint. If it finds one, it starts one FSM Player process to handle the connection and start redirecting the packets toward it. If not, it let the packets pass through so they can reach the real destination on the Internet.

Finally, the endpoint analysis is implemented as a series of Python scripts. One is responsible to process the available network traces and to cluster them together according to the contacted endpoints. The normalization is implemented by a separate tool that dissects the packets, changes the answer of the DNS requests, and replaces the corresponding IPs in the rest of the network traffic.

4.4 Evaluation

In this section we describe the experiments we performed to evaluate Mozzie's ability to model real malware traffic. All the experiments were performed on an Ubuntu 10.10 machine running ScriptGen, Mozzie, and iptables v1.4.4. To perform the live experiments, we ran all samples in a Cuckoo Sandbox [23] running a Windows XP SP3 virtual machine.

4.4.1 System Setup

The goal of our evaluation is to automatically find the minimum number of network traces required to generate a finite state machine that can be used to fully contain the network traffic generated by a given malware sample.

Sample	Category	Containment	Endpoint Normalization	Traces
W32/Virut	IRC Botnet	FULL	NO	15
PHP/Pbot.AN	IRC Botnet	FULL	NO	12
W32/Koobface.EXT	HTTP Botnet	72%	YES	9
W32/Agent.VCRE	Dropper	FULL	NO	23
W32/Agent.XIMX	Dropper	FULL	YES	10

Table 4.1 – Results of the **Offline** learning Experiments

To reach this goal, the first step of our experiments consisted in testing ScriptGen and properly tuning its parameters for the protocols we wanted to model. In fact, in our system we use ScriptGen to model the network behavior of a generic program, but this is not the scenario for which the learning protocol tool was designed in the first place. As we already explained in Section 4.2, the best thresholds of ScriptGen's learning algorithm were experimentally set to the values that were observed to work well (in average) for network worms and remote exploits. However, those thresholds need to be re-computed for different protocols, in particular when moving from a text-based (e.g., HTTP) to a binary format (e.g., RPC).

In the first part of our experiments, we performed a number of tests to learn the optimal parameters for a number of protocols that are commonly used by several malware samples, namely HTTP, IRC, DNS, and SMTP. This step requires the algorithm to be executed several times on the same protocol traces, each time with different parameters [*]. Once the optimal setup was reached, we reused the same values for all the malware samples that used the same protocol. However, in one of the experiments, we discovered that the malware under analysis implemented a custom binary protocol. Since we did not have the ScriptGen configuration for that protocol, we had to re-apply the learning phase for that particular sample traffic.

Even though this operation can take several hours, it is important to note that if the thresholds are not set to their optimal values our technique would still be able to model unknown protocols, even though the system would require an higher number of traces to reach the full containment.

4.4.2 Experiments

Our experiments with real malware can be divided in two groups. In the first case, that we label *"offline"* learning, we use our system to model old traces collected in the past for polymorphic samples. Malware sandboxes normally avoid executing the same sample multiple times, returning the previously computed results when they recognize (usually from the MD5) that a file was already analyzed in the past. However, in case of polymorphic variations, it is possible that the same malware family gets executed several times. Based on this observation, we ex-

∗. Please refer to [127, 128] for a description of the procedure required to set the thresholds

tracted from the Anubis [19] database the network traffic dumps associated to five polymorphic samples that adopted cleartext protocols in their communication.

Our goal was to show that, by using these traces, we can model the network behavior of the malware, and use the extracted FSM to replay and contain the execution of any other polymorphic variation of the same sample.

The results of this first experiment are reported in Table 4.1. The first two columns report the antivirus label and the malware category associated to each sample. The next column reports the success of the experiment, where a `FULL` (100%) value means that full containment was achieved and all the packets were properly replayed. In only one case, for the Koobface malware, our system was not able to successfully model the entire network traffic. The reason is the fast flux approach adopted by Koobface in which both the domain names *and* the IP addresses rotate. This makes it impossible for Mozzie to correctly model the DNS protocol, since there are not two sequences of request/response that look the same in the dataset.

Column 4 shows whether the normalization step was applied to the traffic. As we already explained in Section 4.3, clustering is required to deal with noise in the network traces, most of the time introduced by an anomalous execution of the malware (e.g., due to a network timeout on a web request). On top of that, certain malwares require a normalization phase to properly sanitize the traces from randomization introduced by IP or domain flux techniques.

Finally, the last column of the table shows the number of input traces required to successfully model the traffic. We started the experiment by running Mozzie on a single network trace. We then loaded the extracted model in a virtual machine and used it to try to contain five consecutive executions of another polymorphic variation of the same malware. The reason behind the five runs is that we wanted to be sure that ScriptGen did not return the right message by chance, and that the experiment can be reliably repeated multiple times. If the extracted model was not sufficient to properly "replay" the network conversation, we added one more network trace to the learning pool and repeated the experiment. The number in the fifth column represents the number of network traces required to create a Finite State Machine that achieved full containment (or its best approximation in the case of Koobface) of the malware sample. The results vary between 9 and 23 traces. These numbers may seem large, but it is important to remember that ScriptGen is completely protocol agnostic and that each experiment was performed starting with an empty protocol model. For example, we discovered that 6 traces is the minimum amount required to properly model a DNS request/response exchange (due to the fact that the response has to contain the same request ID field used in the request).

Table 4.2 reports the result of our second group of tests. In this second experiment, we focused on incremental learning, i.e., on analyzing current malware in a sandbox environment each time refining our model of the network traffic. We started by executing the samples 3 times, without attempting to replay the traffic.

Sample	Category	Runs	Containment	Endpoint Normalization
W32/Banload.BFHV	Dropper	23	FULL	NO
W32/Downloader	Dropper	25	FULL	NO
W32/Troj_Generic.AUULE	Ransomware	4	FULL	NO
W32/Obfuscated.X!genr	Backdoor	6	FULL	NO
SCKeylog.ANMB	Keylogger	14	FULL	YES

Table 4.2 – Results of the **Incremental** learning Experiments

Then we created our first model, and started executing the sample in the sandbox with Mozzie acting as a proxy. Whenever the system was not able to contain the traffic, the requests were forwarded to the real servers, and the FSM updated with the new information. The third column reports the number of time each malware has to be analyzed before the model can achieve full containment for five consequent runs.

Overall, we tested 2 IRC botnets, 1 HTTP botnet, 4 droppers, 1 ransomware, 1 backdoor and 1 keylogger. For these samples, we needed a number of network traces ranging from 4 to 25. The first number seems in contradiction with the lower bound we have previously found. The truth is that this particular sample does not use DNS and thus contact the C&C servers directly by using an hardcoded IP address. For all the other malware that generate DNS traffic the number is definitively higher than the lower bound. On average we need 14 traces to be able to build a good traffic model.

Certainly, large malware analysis systems forced to analyze tens or hundreds of thousand of samples per day cannot afford to repeat the tests 14 times. However, such a high number of new samples collected every day is largely due to the common use of polymorphism and packing techniques by malware writers. Therefore, once a FSM is available for one of the samples in the family, any further variation that preserves the behavior of the program does not require any additional training. Our system could help analyzing polymorphic samples for which the required network infrastructure is not available any more, and that nowadays cannot be tested at all. This, as we already described in the introduction, can improve the result of clustering, and can help malware analysts to properly label those samples that do not work anymore at the time of the analysis. Even better, our system could be used to replicate a specific network scenario that is targeted by a malware infection. Recent years have seen the rise of sophisticated attacks targeting specific environments, such as Stuxnet and Duqu [190, 191]. In these cases, the network traces obtained from the targeted network infrastructure (e.g. traces of interaction in a DCS system in a power generation control system) could be used to build a model of the targeted network environment, allowing the analysis of the malware to be successfully performed inside traditional, and safe, sandboxes.

4.5 Limitations

The current prototype of our containment system has several limitations. Some are specific to the way the system has been implemented and some are related to the self-imposed constraints associated to the chosen methodology.

More specifically, we can group the current limitations into three main families:

— The method adopted in this paper is completely protocol-agnostic. While its nature allows us to guarantee our ability to handle custom, undocumented protocols that can be adopted by future malware, it also imposes unnecessary constraints when dealing with simpler and well known protocols such as DNS. We have already seen that our system requires six samples of network interaction to learn how to properly replay a DNS request. The same result could be easily achieved by analyzing only one request, parsing the DNS fields, and extracting the required information in an ad-hoc fashion. However, the goal of this paper was to show how far it is possible to go with a completely generic system. Therefore, our results can be considered as a upper bound, as the system could be easily improved by adding ad-hoc handlers for common and well known protocol interactions.

— Our current prototype is implemented as a network proxy. Even though this approach has some advantages (e.g., it can be easily plugged into any existing sandbox), it makes the analysis of encrypted protocols impossible. However, this is mostly a technical limitation. The same approach could be implemented at the API level, where most of the network traffic is still available in clear text. Most of the malware sandbox environments already hook into the Windows API to extract information about the malware behavior. By adding our system to the hooked network and cryptographic APIs, we could intercept the communication on the host side and achieve full containment also for some encrypted protocols (e.g., the ones based on SSL).

— Our approach is very inefficient when a malware sample exhibits different behaviors independently of the input it receives from the network. For example, if a sample randomly selects the action to perform out of many possible options, Mozzie would require a lot of traces to properly model all possible behaviors. As an extreme case, domain flux techniques (or large pools of domain names like the one described in Section 6.5) cannot be modeled by our system without requiring protocol-aware heuristics, such as handling the DNS interaction by using a custom DNS service.

Chapter 5

Hypervisor Memory Forensics

5.1 Introduction

In this chapter, we present a set of techniques to extend the field of memory forensics toward the analysis of hypervisors and virtual machines. With the increasing adoption of virtualization techniques (both as part of the cloud and in normal desktop environments), we believe that memory forensics will soon play a very important role in many investigations that involve virtual environments.

In some way, the problem of finding a hypervisor is similar to the one of being able to automatically reconstruct information about an operating system in memory, even though that operating system may be completely unknown. The number of commodity hypervisors is limited and, given enough time, it would be possible to analyze all of them and reverse engineer their most relevant data structures, following the same approach used to perform memory forensics of known operating systems. However, custom hypervisors are easy to develop and they are already adopted by many security-related tools [78, 142, 177, 181]. Moreover, malicious hypervisors (so far only proposed as research prototypes [67, 110, 170, 208]) could soon become a reality - thus increasing the urgency of developing the area of virtualization memory forensics.

The main idea behind our approach is that, even though the code and internals of the hypervisors may be unknown, there is still one important piece of information that we can use to pinpoint the presence of a hypervisor. In fact, in order to exploit the virtualization support provided by most of the modern hardware architectures, the processor requires the use of particular data structures to store the information about the execution of each virtual environment. By first finding these data structures and then analyzing their content, we can reconstruct a precise representation of what was running in the system under test.

Starting from this observation, this contribution outlines three main goals. First, we want to extend traditional memory forensic techniques to list the hypervisors present in a physical memory image. As it is the case for traditional operating systems, we also want to extract as much information as possible regarding those

hypervisors, such as their type, location, and the conditions that trigger their behaviors. Second, we want to use the extracted information to reconstruct the address space of each virtual machine. The objective is to be able to transparently support existing memory analysis techniques. For example, if a Windows user is running a second Windows OS inside a virtual machine, thanks to our techniques a memory forensic tool to list the running processes should be able to apply its analysis to either one or the other operating system. Finally, we want to be able to detect cases of nested virtualization, and to properly reconstruct the hierarchy of the hypervisors running in the system.

To summarize, we make the following contributions:

— We are the first to design a forensics framework to analyze hypervisor structures in physical memory dumps.

— We implemented our framework in a tool named Actaeon, consisting of a Volatility plugin, a patch to the Volatility core, and a standalone tool to dump the layout of the Virtual Machine Control Structure (VMCS) in different environments.

— We evaluate our framework on several open source and commercial hypervisors installed in different nested configurations. The results show that our system is able to properly recognize the hypervisors in all the configuration we tested.

5.2 Background

Before presenting our approach for hypervisor memory forensics we need to introduce the Intel virtualization technology and present some background information on the main concepts we will use in the rest of the paper.

5.2.1 Intel VT-x Technology

In 2005, Intel introduced the VT-x Virtualization Technology [95], a set of processor-level features to support virtualization on the x86 architecture. The main goal of VT-x was to reduce the virtualization overhead by moving the implementation of different tasks from software to hardware.

VT-x introduces a new instruction set, called Virtual Machine eXtension (VMX) and it distinguishes two modes of operation: VMX *root* and VMX *non root*. The VMX root operation is intended to run the hypervisor and it is therefore located below “ring 0”. The non root operation is instead used to run the guest operating systems and it is therefore limited in the way it can access hardware resources. Transitions between non root and root modes are called `VMEXIT`, while the transition in the opposite direction are called `VMENTRY`. As part of the VT-x technology, Intel introduced a set of new instructions that are available when the processor is operating in VMX root operation, and modified some of the existing instructions to trap (e.g., to cause a `VMEXIT`) when executed inside a guest OS.

5.2.2 VMCS Layout

VMX transitions are controlled by a data structure called Virtual Machine Control Structure (VMCS). This structure manages the transitions from and to VMX non root operation as well as the processor behavior in VMX non root operation. Each logical processor reserves a special region in memory to contain the VMCS, known as the VMCS region. The hypervisor can directly reference the VMCS through a 64 bit, 4k-aligned physical address stored inside the *VMCS pointer*. This pointer can be accessed using two special instructions (`VMPTRST` and `VMPTRLD`) and the VMCS fields can be configured by the hypervisor through the `VMREAD`, `VMWRITE` and `VMCLEAR` commands.

Theoretically, an hypervisor can maintain multiple VMCSs for each virtual machine, but in practice the number of VMCSs normally matches the number of virtual processors used by the guest VM. The first word of the VMCS region contains a revision identifier that is used to specify which format is used in the rest of the data structure. The second word is the `VMX_ABORT_INDICATOR`, and it is always set to zero unless a VMX abort is generated during a `VMEXIT` operation and the logical processor is switched to shutdown state. The rest of the structure contains the actual VMCS data. Unfortunately, the memory layout (order and offset) of the VMCS fields is not documented and different processors store the information in a different way.

Every field in the VMCS is associated with a 32 bit value, called its *encoding*, that needs to be provided to the `VMREAD/VMWRITE` instructions to specify how the values has to be stored. For this reason, the hypervisor has to use these two instructions and should never access or modify the VMCS data using ordinary memory operations.

The VMCS data is organized into six logical groups: 1) a *guest state area* to store the guest processor state when the hypervisor is executing; 2) a *host state area* to store the processor state of the hypervisor when the guest is executing; 3) a *VM Execution Control Fields* containing information to control the processor behavior in VMX non root operation; 4) *VM Exit Control Fields* that control the VMEXITs; 5) a *VM Entry Control Fields* to control the VMENTRIES; and 6) a *VM Exit Info Fields* that describe the cause and the nature of a VMEXIT.

Each group contains many different fields, but the offset and the alignment of each field is not documented and it is not constant between different Intel processor families [*].

5.2.3 Nested Virtualization

Nested virtualization has been first defined by Popek and Goldberg [83, 164] in 1973. Since then, several implementation has been proposed. In a nested virtualization setting, a guest virtual machine can run another hypervisor that in turn

*. For more information on each VMCS section please refer to the Intel Manual Vol 3B Chapter 20

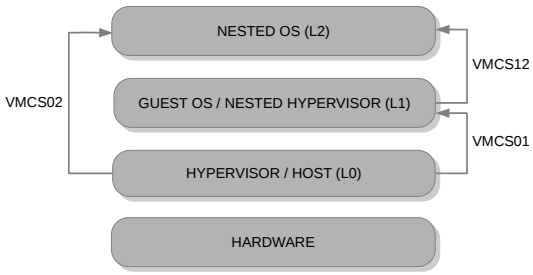

Figure 5.1 – VMCS structures in a Turtle-based nested virtualization setup

can run other virtual machines, thus achieving some form of recursive virtualization. However, since the x86 architecture provides only a single-level architectural support for virtualization, there can only be one and only one hypervisor mode and all the traps, at any given nested level, need to be handled by this hypervisor (the "top" one in the hierarchy). The main consequence is that only a single hypervisor is running at ring -1 and has access to the VMX instructions. For all the other nested hypervisors the VMX instructions have to be emulated by the top hypervisor to provide to the nested hypervisors the illusion of running in root mode.

Because of these limitations, the support for nested virtualization needs to be implemented in the top hypervisor. KVM has been the first x86 virtual machine monitor to fully support nested virtualization using the Turtle technology [43]. For this reason, in the rest of this paper we will use the KVM/Turtle nomenclature when we refer to nested hypervisors. Recent versions of Xen also adopted the same concepts and it is reasonable to think that also proprietary hypervisors (such as VMware and Hyper-V) use similar implementations.

The Turtle architecture is depicted in Figure 5.1. In the example, the top hypervisor (L0) runs a guest operating system inside which a second hypervisor (L1) is installed. Finally, this second hypervisor runs a nested guest operating system (L2). In this case the CPU uses a first VMCS (VMCS01) to control the top hypervisor and its guest. The nested hypervisor has a "fake" VMCS (VMCS12) to mange the interaction with its nested OS (L2). Since this VMCS is not real but it is emulated by the top hypervisor, its layout is not decided by the processor, but can be freely chosen by the hypervisor developers. The two VMCSs are obviously related to each other. For example, in our experiments, we observed that for KVM the VMCS12 Host State Area corresponds to the VMCS01 Guest State Area.

The Turtle approach also adds one more VMCS (VMCS02), that is used by the top hypervisor (L0) to manage the nested OS (L2). In theory, nested virtualization could be implemented without using this additional memory structure. However, all the hypervisors we analyzed in our tests adopted this approach.

Another important aspect that complicates the nested virtualization setup is the memory virtualization. Without nested virtualization, the guest operating system has its own page tables to translate the Guest Virtual Addresses (GVAs) to the Guest Physical Addresses (GPAs). The GPA are then translated by the hypervisor

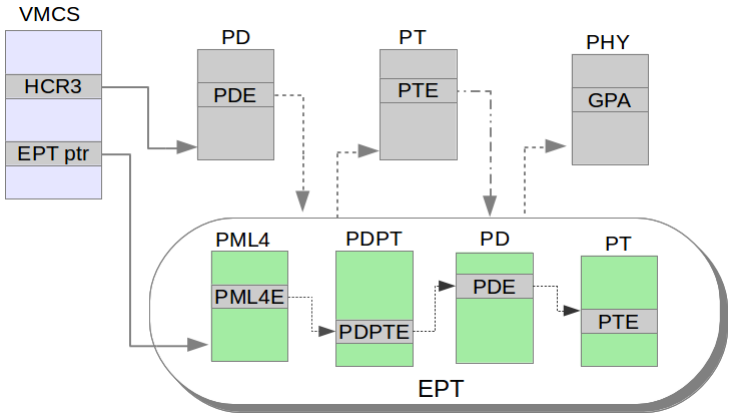

Figure 5.2 – EPT-based Address Translation

to Host Physical Addresses (HPAs) that are pointing to the actual physical pages containing the data. This additional translation can be done either in software (e.g., using shadow page tables [182]) or in hardware (e.g., using the Extended Page Tables (EPT) described later in this section). The introduction of the nested virtualization adds one more layer of translation. In fact, the two dimensional support is no longer enough to handle the translation for nested operating systems. For this reason, Turtle introduced a new technique called multidimensional-paging in which the nested translations (from L2 to L1 in Figure 5.1) are multiplexed into the two available layers.

5.2.4 Extended Page Table

Since the introduction of the *Nehalem* microarchitecture [9], Intel processors adopted an hardware feature, called Extended Page Tables (EPT), to support address translation between GPAs and HPAs. Since the use of this technology greatly alleviated the overhead introduced by memory translation, it quickly replaced the old and slow approach based on shadow pages tables.

When the EPT is enabled, it is marked with a dedicated flag in the *Secondary Based Execution Control Field* in the VMCS structure. This tells the CPU that the EPT mechanism is active and it has to be used to translate the guest physical addresses.

The translation happens through different stages involving four EPT paging structures (namely PML4, PDPT, PD, and PT). These structures are very similar to the ones used for the normal IA-32e address mode translation. If the paging is enabled in the guest operating system the translation starts from the guest paging structures. The PML4 table can be reached by following the corresponding pointer in the VMCS. Then, the GPA is split and used as offset to choose the proper entry at each stage of the walk. The EPT translation process is summarized in Figure 5.2. †

†. For more detail about EPT look at Vol 3B, Chapter 25 Intel Manuals.

5.3 Objectives and Motivations

Our goal is to bring the memory forensic area to the virtualization world. This requires the introduction of new techniques to detect, recognize, and analyze the footprint of hypervisors inside the physical memory. It also requires to support previous techniques, so that existing tools to investigate operating systems and user-space programs could be easily applied to each virtual machine inside a memory image.

Locate Hypervisors in Memory

If an hypervisor is known, locating it in memory could be as simple as looking for a certain pattern of bytes (e.g., by using a code-based signature). Unfortunately, this approach have some practical limitations. In fact, given a snapshot of the physical memory collected during an investigation, one of the main question we want to ask is "Is there *any* hypervisor running on the system?". Even though a signature database could be a fast way to detect well-known products, custom hypervisors are nowadays developed and used in many environments. Moreover, thin hypervisor could also be used for malicious purposes, such as the one described by Rutkowska [170], that is able to install itself in the system and intercept critical operations. Detecting this kind of advanced threats is also going to become a priority for computer forensics in the near future.

For these reasons, we decided to design a **generic** hypervisor detector. In order to be generic, it needs to rely on some specific features that are required by all hypervisors to run. As explained in the previous section, to provide hardware virtualization support, the processor requires certain data structures to be maintained by the hypervisor. For Intel, this structure is called VMCS, while the equivalent for AMD is called VMCB. If we can detect and analyze those structures we could use them as entry points to find all the other components: hypervisors, hosts, and guest virtual machines.

To show the feasibility of our approach, we decided to focus our effort on the Intel architecture. There are two reasons behind this choice. First, Intel largely dominates the market share (83% vs 16% in the second quarter of 2012 [1]). Second, the AMD virtualization structures are fixed and well documented, while Intel adopts a proprietary API to hide the implementation details. Even worse, those details vary between different processor families. Therefore, it provided a much harder scenario to test our techniques.

A limitation of our choice is that our approach can only be applied to hardware assisted hypervisors. Old solutions based on para-virtualization are not supported, since in this case the virtualization is completely implemented in software. However, these solution are becoming less and less popular because of their limitations in terms of performance.

Analysis of Nested Virtualization

Finding the top hypervisor, i.e. the one with full control over the machine, is certainly the main objective of a forensic analysis. But since now most of the commodity hypervisors support nested virtualization, extracting also the hierarchy of nested hypervisors and virtual machines could help an analyst to gain a better understanding of what is running inside the system.

Unfortunately, developing a completely generic and automated algorithm to forensically analyze nested virtualization environments is - in the general case - impossible. In fact, while the top hypervisor has to follow specific architectural constraints, the way it supports nested hypervisors is completely implementation specific. In a nested setup, the top hypervisor has to emulate the VMX instructions, but there are no constraints regarding the location and the format in which it has to store the fields of the nested VMCS. In the best-case scenario, the fields are recorded in a custom VMCS-like structure, that we can reverse engineer in an automated way by using the same technique we use to analyze the layouts of the different Intel processor families. In the worse case, the fields could be stored in complex data structures (such as hash tables) or saved in an encoded form, thus greatly complicating the task of locating them in the memory dump.

Not every hypervisor support nested virtualization (e.g. VirtualBox does not). KVM and Xen implement it using the Turtle [43] approach, and a similar technique to multiplex the inner hypervisors VT-x/EPT into the underlying physical CPU is also used by VMware [32].

By looking for the nested VMCS structure (if known) or by recognizing the VMCS02 of a Turtle-like environment (as presented in Figure 5.1 and explained in details in Section 6.4), we can provide an extensible support to reconstruct the hierarchy of nested virtualization.

Virtual Machine Forensic Introspection

Once a forensic analyst is able to list the hypervisors and virtual machines in a memory dump, the next step is to allow her to run all her memory forensic tools on each virtual machine. For example, the Volatility memory forensic framework ships with over 60 commands implementing different kinds of analysis - and many more are available through third-party plugins. Unfortunately, in presence of virtualization, all these commands can only be applied to the host virtual machine. In fact, the address spaces of the other VMs require to be extracted and translated from guest to host physical addresses.

The goal of our introspection analysis is to parse the hypervisor information, locate the tables used by the EPT, and use them to provide a transparent mechanism to translate the address space of each VM.

5.4 System Design

Our hypervisor analysis technique consists of three different phases: memory scanning, data structure validation, and hierarchy analysis. The Memory Scanner takes as input a memory dump and the database of the known VMCS layouts (i.e., the offset of each field in the VMCS memory area) and outputs a number of candidate VMCS. Since the checks performed by the scanner can produce false positives, in the second phase each structure is validated by analyzing the corresponding page table. The final phase of our approach is the hierarchy analysis, in which the validated VMCSs are analyzed to find the relationships among the different hypervisors running on the machine.

In the following sections we will describe in details the algorithms that we designed to perform each phase of our analysis.

5.4.1 Memory Scanner

The goal of the memory scanner is to scan a physical memory image looking for data structures that can represent a VMCS. In order to do that, we need two types of information: the memory layout of the structure, and a set of constraints on the values of its fields that we can use to identify possible candidates. The VMCS contains over 140 different fields, most of which can assume arbitrary values or they can be easily obfuscated by a malicious hypervisors. The memory scanner can tolerate false positives (that are later removed by the validation routine) but we want to avoid any false negative that could result in a missed hypervisor. Therefore we designed our scanner to focus only on few selected fields:

— `Revision ID`: It is the identifier that determines the layout of the rest of the structure. For the VMCS of the top hypervisor, this field has to match the value of the `IA32_VMX_BASIC MSR` register of the machine on which the image was acquired (and that changes between different micro-architecture). In case of nested virtualization, the revision ID of the VMCS12 is chosen by the top hypervisor. The `Revision ID` is always the first word of the VMCS data structure.

— `VMX ABORT INDICATOR`: This is the VMX abort indicator and its value has to be zero. The field is the second entry of the VMCS area.

— `VmcsLinkPointerCheck`: The values of this field consists of two consecutive words that, according to the Intel manual, should always be set to `0xffffffff`. The position of this field is not fixed.

— `Host_CR4`: This field contains the host CR4 register. Its 13th bit indicates if the VMX is enabled or not. The position of this field is not fixed.

To be sure that our choice is robust against evasions, we implemented a simple hypervisor in which we tried to obfuscate those fields during the guest operation and re-store them only when the hypervisor is running, a similar approach is described in [75]. This would simulate what a malicious hypervisor could do in order to hide the VMCS and avoid being detected by our forensic technique. In

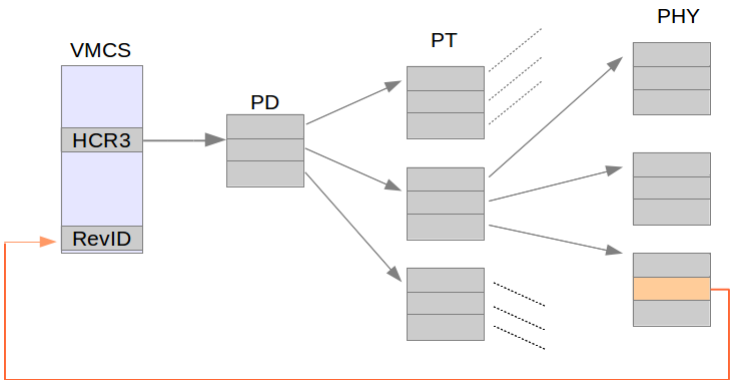

Figure 5.3 – Self-referential Validation Technique

our experiments, any change on the values of the previous five fields produced a system crash, with the only exception of the Revision ID itself. For this reason, we keep the revision ID only as a key in the VMCS database, but we do not check its value in the scanning phase.

The memory scanner first extracts the known VMCS layouts from the database and then it scans the memory looking for pages containing the aforementioned values at the offsets defined by the layout. Whenever a match is found, the candidate VMCS is passed over to the validation step.

5.4.2 VMCS Validation

Our validation algorithm is based on a simple observation. Since the `HOST_CR3` field needs to point to the page table that is used by the processor to translate the hypervisor addresses, that table should also contain the mapping from virtual to physical address for the page containing the VMCS itself. We call this mechanism self-referential validation.

For every candidate VMCS, we first extract the `HOST_CR3` field and we assume that it points to a valid page table structure. Unfortunately, a page table can be traversed only by starting from a virtual address to find the corresponding physical one, but not vice-versa. In our case, since we only know the physical address of the candidate VMCS, we need to perform the opposite operation. For this reason, our validator walks the entire page tables (i.e., it tries to follow every entry listed in them) and creates a tree representation where the leaves represent the mapped physical memory pages and the different levels of the tree represent the intermediate points of the translation algorithm (i.e., the page directory, and the page tables).

This structure has a double purpose. First, it serves as a way to validate a candidate VMCS, by checking that one of the leaves points to the VMCS itself (see Figure 5.3). If this check fails, the VMCS is discarded as a false positive. Second, if the validation succeeded, the tree can be used to map all the memory pages

that were reserved by the hypervisor. This could be useful in case of malicious hypervisors that need an in-depth analysis after being discovered.

It is important to note that the accuracy of our validation technique leverages on the assumption that is extremely unlikely that such circular relationship can appear by chance in a memory image.

5.4.3 Reverse Engineering The VMCS Layout

The previous analysis steps are based on the assumption that our database contains the required VMCS layout information. However, as we already mentioned in the previous sections, the Intel architecture does not specify a fix layout, but provides instead an API to read and write each value, independently from its position.

In our study we noticed that each processor micro-architecture defines different offsets for the VMCS fields. Since we need these offsets to perform our analysis, we design and implement a small hypervisor-based tool to extract them from a live system.

More in detail, our algorithm considers the processors microcode as a black box and it works as follows. In the first step, we allocate a VMCS memory region and we fill the corresponding page with a 16 bit-long incremental counter. At this point the VMCS region contains a sequence of progressive numbers ranging from 0 to 2048, each representing its own offset into the VMCS area. Then, we perform a sequence of `VMREAD` operations, one for each field in the VMCS. As a result, the processor retrieves the field from the right offset inside the VMCS page and returns its value (in our case the counter that specifies the field location).

The same technique can also be used to dump the layout of nested VMCSs. However, since in this case our tool would run as a nested hypervisor, the top hypervisor could implement a protection mechanism to prevent write access to the VMCS region (as done by VMware), thus preventing our technique to work. In this case we adopt the opposite, but much slower, approach of writing each field with a `VMWRITE` and then scan the memory for the written value.

5.4.4 Virtualization Hierarchy Analysis

If our previous techniques detect and validate more then one VMCS, we need to distinguish between several possibilities, depending whether the VMCS represent parallel guests (i.e., a single hypervisor running multiple virtual machines), nested guests (i.e, an hypervisor running a machine the runs another hypervisor), or a combination of the previous ones.

Moreover, if we assume one virtual CPU per virtual machine, we can have three different nested virtualization scenarios: Turtle approach and known nested VMCS layout (three VMCSs found), Turtle approach and unknown nested layout (two VMCSs found), and non-Turtle approach and known layout (two or more VMCSs found).

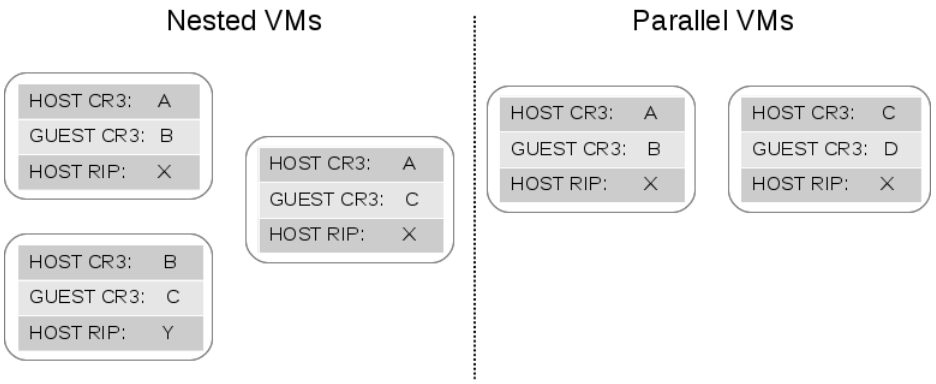

Figure 5.4 – Comparison between different VMCS fields in nested and parallel configurations

In the first two cases (the only ones we could test in our experiments since all the hypervisors in our tests adopted the Turtle approach), we can infer the hierarchy between the hypervisors and distinguish between parallel and nested VMs by comparing the values of three fields: the `GUEST CR3`, the `HOST CR3`, and the `HOST RIP`. The first two fields represent the CR3 for the guest and for the hypervisor. The third is the pointer to the hypervisor entry point, i.e., to the first instruction to execute when the CPU transfer control to the hypervisor.

Figure 5.4 show a comparison of the values of these three fields in a parallel and nested configurations. As the diagram shows, in a nested setup we have two different hypervisors (represented by the two different `HOST RIP` addresses) while for parallel virtual machine the hypervisor is the same (same value of `HOST RIP`). Moreover, by comparing the `GUEST CR3` and `HOST CR3` values we can distinguish among VMCS01, VMCS02, and VMCS12 in a nested virtualization setup. More precisely, the VMCS01 and VMCS02 share the same `HOST CR3`, while the `HOST CR3` of the VMCS12 has to match the `GUEST CR3` of the VMCS01.

Finally, in the third scenario in which the nested virtualization is not implemented following the Turtle approach (possible in theory but something we never observed in our experiments), the previous heuristics may not work. However, also in this case we can still tell that a VMCS belongs to a nested hypervisor if its layout matches the one of a known nested VMCS (e.g., the one emulated by KVM).

5.4.5 Virtual Machine Introspection

The last component of our system is the algorithm to extract the EPT tables and to provide support for the memory analysis of virtual machines. In this case the algorithm is straightforward. First, we extract the pointer to the EPT from the VMCS of the machine we want to analyze (see Figure 5.2). Then, we simulate the EPT translation by programmatically walking through the PML4, PDPT, PD, and PT tables for each address that need to be translated.

5.4.6 System Implementation

We implemented the previously described techniques in an open source tool called Actaeon. Actaeon consists of three components: a standalone VMCS layout Extractor derived from HyperDbg [78], an hypervisor Memory Analysis plugin for the Volatility framework, and a patch for the Volatility core to provide a transparent mechanism to analyze the virtual machines address spaces. The tool, along with a number of datasets and usage examples, can be downloaded from `http://s3.eurecom.fr/tools/actaeon`.

VMCS Layout Extractor

This component is designed to extract and save into a database the exact layout of a VMCS, by implementing the reverse engineering algorithm described above. The tool is implemented as a small custom hypervisor that re-uses the initialization code of HyperDbg, to which it adds around 200 lines of C code to implement the custom checks to identify the layout of the VMCS.

Hyper-ls

This component is implemented as a Python plugin for the Volatility framework, and it consists of around 1,300 lines of code. Its goal is to scan the memory image to extract the candidate VMCSs, run our validation algorithm to filter out the false positives, and analyze the remaining structures to extract the details about the corresponding hypervisors.

The tool is currently able to parse all the fields of the VMCS and to properly interpret them and print them in a readable form. For example, our plugin can show which physical devices and which events are trapped by the hypervisor, the pointer to the hypervisor code, the Host and Guest CR3, and all the saved CPU registers for the host and guest systems.

The `hyperls` plugin can also print a summary of the hierarchy between the different hypervisors and virtual machines. For each VM, it also reports the pointer to the corresponding EPT, required to further inspect their content.

Virtual Machine Introspection Patch

An important functionality performed by Acteon is to provide a transparent mechanism for the Volatility framework to analyze each Virtual Machine address space. In order to provide such functionality, Acteon provides a patch for the Volatility core to add one command-line parameter (that the user can use to specify in which virtual machine he wants to run the analysis) and to modify the APIs used for address translations by inserting an additional layer based on the EPT tables. The patch is currently implemented in 250 lines of Python code.

5.5 Evaluation

The goal of our experiments is to evaluate the accuracy and reliability of our techniques in locating hypervisors inside physical memory dumps, access their private data, reconstruct the hierarchy in case of nested virtualization, and provide the support for other memory forensic techniques to inspect the guest operating systems. All the experiments have been performed on an Intel Core 2 Duo P8600 and an Intel Core i5-2500 machines running the Ubuntu Linux 12.10 32bit operating system and with one virtual processor per guest.

5.5.1 Forensic Memory Acquisition

The first step of our experiments consisted in the acquisition of complete snapshots of the physical memory on a computer running a number of different hypervisor configurations.

As we already mentioned in Section 5.1, this turned out to be a challenging task. In fact, even though a large number of memory imaging solution exists on the market, the vast majority adopt software-based techniques that uses kernel modules to acquire the memory from the operating system point of view. These approaches have not been designed to work in a virtualization environment where the OS does not have a complete view of the system memory. In fact, if the virtual machine monitor is protecting its own pages, the memory image collected from the host operating system does not contain the pages of the hypervisor. To overcome this limitation, whenever a software approach was not able to properly capture the memory, we resorted to a hardware-based solution. In particular, we used a PCI Firewire card with a Texas Instrument Chipset, and the Inception [5] tool to dump the memory through a DMA attack [160]. In this case, we had to disable the Intel VT-d support from the BIOS, to prevent the IOMMU from blocking the DMA attack.

The main drawback of using the Firewire acquisition is that in our experiments it was quite unstable, often requiring several consecutive attempts before we could obtain a correct dump. Moreover, it is worth noting that in theory even a DMA-based approach is not completely reliable. In 2007 Joanna Rutkowska showed the feasibility of attacks against hardware-based RAM acquistion [171]. The presented attacks are based on the modification of the processor's NorthBridge memory map to denial of service the acquisition tool or to hide some portions of the physical memory. However, we are not aware of any hypervisor that uses these techniques to tamper with the memory acquisition process.

Today, the best solution to acquire a complete system memory in presence of an hypervisor would be to use an acquisition tool implemented in the SMM (therefore running at higher privileges than the hypervisor itself), as proposed by A. Reina et al. [167]. Unfortunately, we were not able to find any tool of this kind available on the Internet.

Hypervisor	Guests	Candidate VMCS	Validated VMCS
HyperDbg	1	1	1
KVM	2	4	2
Xen	2	3	2
VirtualBox	1	2	1
VMware	3	3	3

Table 5.1 – Single Hypervisor Detection

5.5.2 System Validation

The first step of our experiments was to perform a number of checks to ensure that our memory acquisition process was correct and that our memory forensic techniques were properly implemented.

In the first test, we wrote a simple program that stored a set of variables with known values and we run it in the system under test. We also added a small kernel driver to translate the program host virtual addresses to host physical addresses and we used these physical addresses as offset in the memory image to read the variable and verify their values.

The second test was designed to assess the correctness of the VMCS layout. In this case we instrumented three open source hypervisors to intercept every VMCS allocation and print both its virtual and physical addresses. These values were then compared with the output of our Volatility plugin to verify its correctness. We also used our instrumented hypervisors to print the content of all the VMCS fields and verify that their values matched the ones we extracted from the memory image using our tool.

Our final test was designed to test the virtual machine address space reconstruction through the EPT memory structures. The test was implemented by instrumenting existing hypervisors code and by installing a kernel debugger in the guest operating systems to follow every step of the address translation process. The goal was to verify that our introspection module was able to properly walk the EPT table and translate every address.

Once we verify the accuracy of our acquisition and implementation we started the real experiments.

5.5.3 Single-Hypervisor Detection

In this experiment we ran the `hyperls` plugin to analyze a memory image containing a single hypervisor.

We tested our plugin on three open source hypervisors (KVM 3.6.0, Xen 4.2.0, and VirtualBox 4.2.6), one commercial hypervisor (VMware Workstation 9.0), and one ad-hoc hypervisor realized for debugging purposes (HyperDbg). The results are summarized on Table 5.1. We run the different hypervisors with a variable number of guests (between 1 and 4 virtual machines). The number of candidate

Top Hypervisor	Nested Hypervisor	VMCS Detection	Hierarchy Inference
KVM	HyperDbg	✓	✓
	KVM	✓	✓
XEN	KVM	✓	✓
	XEN	✓	✓
VMware	HyperDbg	✓	✓
	KVM	✓	✓
	VirtualBox	✓	✓
	VMware	✓	✓

Table 5.2 – Detection of Nested Virtualization

VMCS found by the memory scanner algorithm is reported in the third column, while the number of validated ones is reported in the last column. In all the experiments our tool was able to detect the running hypervisors and all the virtual machines with no false positives.

The performance of our system are comparable with other offline memory forensic tools. In our experiment, the average time to scan a 4GB memory image to find the candidate VMCS structures was 13.83 seconds. The validation time largely depends on the number of matches, with an average of 51.36 seconds in our tests (all offline analysis performed on an Intel Xeon L5420 (2.50Ghz) with 4GB RAM).

In the second experiment, we chose a sample of virtual machines from the previous test and we manually inspect them by running several Volatility commands (e.g., to list processes and kernel drivers). In all cases, our patch was able to transparently extract the EPT tables and provide the address translation required to access the virtual machine address space.

5.5.4 Nested Virtualization Detection

In the final set of experiments we tested our techniques on memory images containing cases of nested virtualization. This task is more complex due to the implementation specific nature of the nested virtualization. First of all, only three of the five hypervisors we tested supported this technology. Moreover, not all combinations were possible because of the way the VMX instructions were emulated by the top hypervisor. This turned out to be crucial for the nested hypervisor to work properly, since an imperfect implementation would break the equivalence principle and allow the nested hypervisor to detect that it is not running on bare metal. For example, VMware refuses to run under KVM, while Xen and VirtualBox under KVM start but without any hardware virtualization support.

Because of these limitations we were able to set up eight different nested virtualization installations (summarized in Table 5.2). In all the cases, `hyperls` was

able to detect and validate all the three VMCS structures (VMCS01, VMCS02, and VMCS12) and to infer the correct hierarchy between the different hypervisors.

Chapter 6

Analysis of ROP Chains

6.1 Introduction

In this chapter, we present a set of techniques to analyze complex ROP chains. First, we identify and discuss the main challenges that make it very difficult to reverse engineer code implemented using ROP. Second, we propose an emulation-based framework to dissect, reconstruct, and simplify ROP chains. Finally, we test our tool on the most complex example available to date: a ROP rookit containing four separate chains, two of them dynamically generated at runtime.

Traditional reverse engineering relies on a wide range of tools that have been perfected over the years, such as debuggers, disassemblers, and decompilers. Unfortunately, all these products were designed for "EIP-based" programming and are of very little use to analyze stack-based return oriented programming payloads. This is the problem we address in this chapter. Specifically, two main observations motivate our work: the lack of public tools to analyze in-depth ROP payloads and the fact that ROP chains are growing both in size and in complexity. Moreover, all the memory forensic techniques proposed so far try to identify injected code and do not consider *code reuse* strategies. This myopia considerably weaken the analyses.

To tackle these problems, we propose ROPMEMU – a framework for the automated analysis of ROP chains. We assume that, using existing techniques [163, 185], a forensic investigator discovers a ROP chain in system memory and she needs to investigate its behavior. At first glance, the problem may seem trivial: it would be enough to dump the memory region containing the ROP chain, reconstruct the entire code by appending the instructions contained in each gadget, and then analyze it like any other sequence of assembly instructions. However, in practice, things are not simple and in this paper we show that this procedure is in fact very complex and requires a number of dedicated tools and techniques.

ROPMEMU leverages techniques from the fields of memory forensics, emulation, multi-path execution, and compiler transformations to analyze complex ROP chains and recover their precise control flow graph. Moreover, by using a

novel multi-path emulation, our system is also able to reconstruct chains which are dynamically-generated at runtime, allowing an analyst to capture the behavior of the most complex ROP that can be encountered in the wild.

To summarize, we present the following contribution:

— We discuss a number of challenges that need to be addressed to reverse engineer code implemented using return oriented programming. This goes far beyond what was observed in the past in simple exploits and what was discussed in previous papers.

— We present the first framework to dissect, reconstruct, and simplify complex ROP chains.

— We tested our tool with the most complex case proposed so far: a ROP rootkit containing chains with a total of 215913 gadgets.

6.2 Background

In this section, we provide the technical background required to understand the remaining part of the paper. We first introduce the return oriented programming paradigm, both from the point of view of exploitation and of the available analysis techniques. We then provide an overview of the current trends and evolution of rootkit technologies as well as the recently-proposed concept of an ROP rootkit. Finally, we introduce in more detail a real example of ROP rootkit (proposed by Vogl et al. [199]) that we will use as a case study throughout the rest of the paper.

6.2.1 ROP

Security countermeasures introduced in the last decade in modern operating systems forced attackers to adapt and find new ways to exploit programs. To overcome hardware defenses – such as the *no-execute* bit (NX) in PAE and IA-32e modes on Intel processors, software protections trying to emulate the NX bit behavior [184, 195, 196], and code signing [2, 7, 8] techniques – offensive researchers proposed several forms of the so-called *code reuse* attack [153, 176, 179]. Over the years, *code reuse* attacks have been ported to different architectures [52, 113] and have evolved in a multitude of different techniques, such as return oriented programming without returns [57], jump return oriented programming [48], blind ROP [47], and sigreturn oriented programming [49].

In particular, ROP is one of the most prevalent and widespread techniques adopted in the majority of the exploits observed in the wild. It is a particular instance of *code reuse* attack in which the attacker uses instructions already present in memory and chains them together to perform arbitrary computation. A single block of assembly instructions terminated by a `ret` (in its most traditional form) is called a *gadget*. A sequence of gadgets is then connected to form a *ROP chain* by putting their addresses on the stack and leveraging the `ret` instruction to return from one gadget to the next one.

ROP Analysis

So far, ROP was mainly used by exploits to disable the protection enforced by the NX bit and then execute normal shellcode. In these cases, the ROP chain is generally very short, as its only goal is to invoke functions (e.g., `VirtualProtect` or `mprotect`) to change the page permissions of the memory containing the shellcode. As a result, the vast majority of ROP chains are straight sequences of instructions without any branch or complex control flow.

There exists a countless number of offensive tools to simplify the creation of ROP chains – ranging from simple tools [36, 61, 106, 174] able to disassemble binaries, find gadgets and group them together – to more advanced tools [35, 154, 156] that use constraint solvers, intermediate languages and even emulators [155] to automate the chain creation as much as possible. Unfortunately, because of the simple form of the existing ROP chains, to date there are no public tools to analyze ROP payloads. In fact, the analysis of ROP payloads was so far purely a manual process in which the analyst dissects the binary to find the chain and then manually de-obfuscates it to understand its behavior.

More recently, researchers and malware writers discovered that return oriented programming is not only a useful technique to run exploits, but it also provides a very effective way to *hide* the execution of malicious functionality. In fact, since ROP allows the implementation of new functionality by reusing existing sequences of instructions, it makes the malicious code much more complex to identify, isolate and analyze. As part of this emerging phenomenon, chains have started to contain complex application logic, therefore becoming much longer and much more complex. As a first example, malware samples have adopted ROP payloads during the first stage of the infection. For instance, immediately after the exploitation, the ROP payload was used to implement a simple dropper/downloader that fetches and runs the second stage [99]. Even more worrying, in 2014 Vogl et al. [199] presented the first complete example of a rootkit implemented in ROP. This opens a new era for malicious code execution, and calls for a new set of tools and techniques to perform its analysis.

6.2.2 Rootkits

Rootkits are malicious software designed to gain persistent, *stealth* access to a compromised machine. In the last few years, rootkit technology has been rapidly evolving and increasing in sophistication. In order to conceal their presence and information, modern rootkits typically run at ring 0. This places the attacker at the same level as the OS kernel, so that the rootkit can undermine the security of the operating system and, potentially, remain undetectable for a long time. Several defensive mechanisms have been proposed to address this issue, but, unfortunately, ring 0 rootkits are still a severe threat. Offensive researchers have also investigated further possible ways to subvert the operating system security model moving

deeper in the execution stack: prototypes exist for virtualization rootkits (ring -1) [70, 103, 111], SMM rootkits (ring -2) [77] and Intel ME rootkits (ring -3) [159]. The trend is to be a level lower than the defensive monitor in order to lie hidden on the compromised system.

Fortunately, all rootkits share a common weakness: they need to load their code into the running system. Modern countermeasures, such as secure boot and code signing, significantly hinder this process and make traditional attacker techniques no more effective against recent systems. To bypass these protection mechanisms, malware authors can use the same techniques adopted by exploit writers. Return Oriented Programming (ROP) is a well-known data-only exploitation technique and can be used by the rootkit authors to bypass all the modern protections. A completely ROP rootkit was first theorized by Hund et al [93] in 2009. They proposed a proof of concept with several limitations. First of all, the malware has to exploit the kernel vulnerability repeatedly to execute arbitrary ROP payloads (e.g., to hide system processes). Second, this initial rootkit had no persistence at all – making it of little use in practice. In 2014, Vogl et al. [199] succeeded in making a ROP rootkit persistent and presented an open-source POC of their creation. In particular, the authors have shown how it is possible to perform hooking without injecting a single line of code in the kernel. In this case, the malware has to exploit the vulnerability only once to escalate privileges and trigger the persistent ROP payload.

6.2.3 Chuck

Chuck is the name of the persistent ROP rootkit proposed by Vogl et al [199], the only public example of this kind to date.

It comprises four ROP chains: one persistent in memory and the other three dynamically generated at runtime. The first chain is the *initialization* chain and it is executed only once, the first time the kernel vulnerability is exploited (in this particular case CVE-2013-2094 [104]). This chain sets the hooks in the system, sets up the switching mechanism based on the `sysenter` instruction using the MSR registers 0x175 (`IA32_SYSENTER_ESP`) and 0x176 (`IA32_SYSENTER_EIP`), the global state of the rootkit, a memory region to deal with multiple invocations, and finally it copies the second chain – the so-called *copy chain*. The *copy chain* is the persistent ROP chain, and it is invoked every time a hook is triggered. First, it saves all the general purpose registers in the global state. Second, it creates and copies in memory a dynamic chain for each invocation of the hook. This third chain is called the *dispatcher chain*. The *dispatcher chain* is necessary to deal with hook invocations by multiple threads. The goal of this chain is to create a final, ad-hoc payload. The *payload* chain contains the core functionality of the rootkit and, at the end of its execution, it restores the original registers to continue normal kernel execution.

The complexity of these ROP chains is considerably high for several reasons. First, the size of a single chain is huge compared to the chains shipped with or-

dinary exploits. For instance the *copy chain* contains over 180k gadgets. Second, these chains have a non-linear control flow logic – making their analysis very complex. Third, the presence of dynamically generated chains make this example similar to a multi-stage packed malware, limiting the applicability of static analysis. Finally, the four chains compose a real kernel rootkit and thus the analyst has to deal with kernel issues such as privileged instructions and interrupts.

6.3 ROP Analysis

To date, the complexity of ROP analysis has been completely underestimated. Few studies have focused on this problem, mainly taking simplistic approaches applied only to small examples. The first issue that an analyst may encounter when dealing with ROP chains is the fact that they are hard to locate in the first place. Since no code is injected in a ROP-based attack, finding the entry point of the chain can be difficult – especially when the input is the entire system memory. Two previous studies have proposed solutions for this problem [163, 185] and therefore we will build on top of them for the rest of our paper. Focusing more on the real analysis (i.e., on what needs to be done *after* a chain has been located) we identify seven main challenges:

[C1] Verbosity – the majority of ROP gadgets contain spurious instructions. For example, a gadget intended to increment `eax` may also pop a value from the stack before hitting the `ret` instruction that triggers the next gadget in the chain. Moreover, the code of a ROP chain contains a large percentage of return or other indirect control flow instructions, whose only goal is to connect together all the other gadgets. These are only few examples of why ROP code is very verbose and contains a large fraction of dead code that makes it harder for analysts to understand it. However, this is probably the simplest problem to solve as many transformations proposed in the compiler literature already exist to simplify the code.

[C2] Stack-Based Instruction Chaining – the most obvious difference between a ROP chain and a normal program is that in a chain the instructions are not consecutive in memory, rather they are grouped in small gadgets connected together by indirect control flow instructions. So, what in a normal program could be a single block of 50 instructions, in a ROP chain can be split into more than 40 blocks chained by `ret` instructions.

At a first glance, this problem may seem trivial to solve. Since the addresses of each gadget in the chain are saved on the stack, one might think that it would be easy to automatically retrieve them, collect the corresponding pieces of code, and replace the entire chain with a single sequence of instructions. However, the stack-based instruction chaining can introduce subtle side effects that are hard to identify with a simple static analysis approach. For instance, since the sequence of gadgets is saved on the stack, but the code of each gadget also interacts with the stack (to retrieve parameters or just because of spurious instructions), in order

to correctly identify the addresses of each gadget it is necessary to emulate every single instruction in the code.

[C3] Lack of Immediate Values – another difference between normal code and ROP chains is the fact that chains are typically constructed with "generic" gadgets (such as "store an arbitrary value in the `rax` register") that operate on parameters which are also stored on the stack. As a result, the vast majority of immediate values that are assigned to registers are interleaved on the stack with the gadget addresses. Being able to restore them back in their original position also requires an emulator.

[C4] Conditional Branches – in a ROP chain, a branch condition implies a change in the stack pointer instead of a more traditional change in the instruction pointer. This means that a simple conditional jump may be encoded with dozens of different instructions spanning multiple gadgets (e.g., to set the flag register according to the required condition, test its value, and conditionally increment the `esp` register). To translate the chain into more readable code, it is therefore necessary to identifying these patterns based on their semantics and replace them with single branch instructions.

[C5] Return to Functions – function calls are typically implemented in ROP as simple return to the functions entry point. However, since normal gadgets are also often extracted from code located inside libraries, it is hard to distinguish a function call from another gadget. As it is in the case of statically linked binaries, the lack of information on external library calls can make the reverse engineering process much more tedious and complicated.

[C6] Dynamically Generated Chains – the instructions of normal programs are typically located in a read-only section of the executable. Dynamically modified code is common in malware (e.g., as a result of packing) and, in fact, this severely limits the ability to perform static analysis on malicious code and considerably slows down the reverse engineering process. On the contrary, ROP chains are located on the stack, and it is therefore simple to use gadgets to prepare the execution of other gadgets in the future. This dynamicity results in the fact that it is not necessary for the entire chain to reside in memory at the same time.

[C7] Stop Condition – in this paper we assume that the analyst is able to locate the beginning of a ROP chain in memory. However, since an emulator is needed to analyze its content, it is important to also have a *termination condition* to decide when all the gadgets have been extracted and the emulation process can be stopped. The fact that complex ROP chains can invoke functions (which in turn may invoke other functions) interleaved with normal gadgets, and the fact that a chain can dynamically generate another chain in a different part of the memory, make this problem very hard to solve in the general case.

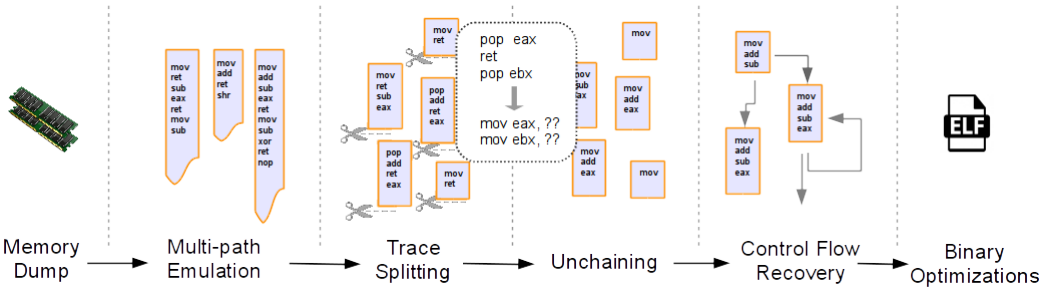

Figure 6.1 – ROPMEMU Framework Architecture

6.3.1 Implications

The previous seven challenges have several important implications for the analysis of ROP chains. Previous works only proposed partial solutions. For instance, Lu et al. [139] and Yadegari et al. [203] identified a number of code transformations to handle **[C1]**. Moreover, Stancill et al. [185] and Lu et al. [139] used simple heuristics to follow the value of the stack pointer, thus partially addressing **[C2]** and **[C3]**. However, previous heuristics only applied to `ret`-based ROP chains, and were unable to follow indirect calls and jump instructions. Sadly, **[C4-7]** have never been mentioned before, probably because only in the past two years ROP chains have become complex enough to raise these points.

As it is better explained in Section 6.4, to fully address **[C2]**,**[C3]**, and **[C6]** it is necessary to emulate all the instructions and keep a shadow copy of the memory content. Moreover, a solution based on *multi-path emulation* is required to explore each path in the chain and retrieve its entire code. In turn, this approach requires the system to implement heuristics to detect the presence of branch instructions (**[C4]**). Finally, while recognizing the functions (**[C5]**) can be addressed by using symbols information extracted from libraries and kernel functions, the presence of system calls is a major obstacle for an emulator because their return values cannot be predicted with static analysis. Functions are not the only issue when using an emulator: precise heuristics for the stopping condition (**C7**) are also required and (as better explained in the next section) hard to implement.

This short discussion emphasizes how ROP analysis is in fact a multi-faced problem whose solution requires a combination of sophisticated techniques.

6.4 Design

The ROPMEMU framework adopts a set of different techniques to analyze ROP chains and reconstruct their equivalent code in a form that can be analyzed by traditional reverse engineering tools. In particular, it is based on *memory forensics* (as its input is a physical memory dump), *code emulation* (to faithfully rebuild the original ROP chain), *multi-path execution* (to extract the ROP chain payload),

CFG recovery (to rebuild the original control flow), and a number of *compiler transformations* (to simplify the final instructions of the ROP chain).

The framework is divided in different components that interact as shown in Figure 6.1 in five main analysis phases:

— **Multipath Emulation** - This step emulates the assembly instructions that compose the ROP chain. This is the only way to rebuild the exact instance of the running chain at the time of the dump. All the possible branches are explored and an independent trace (annotated with the values of registers and memory) is generated for each execution path (**C2** and **C6**). The emulator is also designed to recognize a number of returns-to-library functions, skip over their body, and simulate their execution by generating dummy data and return values (**C4**).

— **Trace Splitting** - This phase consists of analyzing all the traces generated by the emulator, remove the repetitions, and extract the unique blocks of code.

— **Unchaining** - This phase applies a number of assembly transformations to simplify each ROP trace by removing the connections between gadgets and merging the content of consecutive gadgets in a single basic block. This step is also responsible to remove immediate values from the stack and assign them to the corresponding registers (**C2** and **C3**).

— **CFG recovery** - This pass merges all the code blocks in a single program, recovering the original control flow graph of the ROP chain. This phase comprises two steps. In the first one, the traces are merged in a single graph-based representation. The second step translates the graph into a real `x86` program by identifying the instructions associated to the branch conditions and by replacing them with more traditional EIP-based conditional jumps (**C4**). At the end of this phase, the program is saved in an ELF file, to allow traditional reverse engineering tools (e.g., IDA Pro) to operate on it.

— **Binary optimization** - In the final step, we apply known compiler transformations to further simplify the assembly code in the ELF file. For instance, this phase removes dead instructions in the gadgets and generates a clean and higly optimized version of the payload (**C1**).

In the rest of the section, we introduce each phase in detail and we describe how each of them have been implemented in our system.

6.4.1 Chain Discovery

Finding ROP chains in a physical memory dump is not a trivial task. However, two solutions have already been proposed in the literature [163, 185] for this problem. Therefore, for the sake of simplicity, in this paper we assume that the analyst is provided with an image of the memory and an entry point of the first ROP chain.

Our case study was complicated by the fact that only one of the chains is persistent (the *Copy Chain*), while the other ones are generated on the fly depending

on the system's state and therefore their content is only available in memory for few milliseconds. As a consequence, it is unrealistic to require a snapshot of the memory containing all ROP chains – and therefore their content needs to be reconstructed by our system.

6.4.2 Emulation

The emulation phase is the core of our analysis framework. Its role is to "follow" the execution of each gadget to keep an updated position of the stack pointer and of the content of the memory.

A Turing complete ROP chain can be obtained by re-using a limited number of gadgets [179]. Therefore, also very complex ROP programs often include a very small number of *unique* assembly instructions. For instance, the ROP rootkit we used for our experiments contained only 16 mnemonics. For this reason, in our prototype we decided to implement a small custom emulator from scratch. Our emulator supports both `x86-32` and `x86-64` architectures and updates the state of the CPU (registers and flags) and of the memory after each instruction. For a more comprehensive approach, S2E [59] can be used to provide a full system emulation on top of Qemu [42]. However, this would considerably complicate the setup and deployment required by the system. Therefore, we opted for a custom solution that gave us more flexibility and a smaller footprint.

At the beginning of the emulation phase, the initial state of the virtual CPU is set to zero by resetting the content of all registers except for the instruction pointer and the stack pointer (whose initial values need to be provided as input for our analysis). The emulator is then implemented as a Volatility [12] plugin to simplify the interaction with the memory dump by leveraging the Volatility APIs.

Execution Modes

At the end of the emulation a JSON trace is generated containing the CPU state for each assembly instruction. Depending on the complexity of the ROP chain, the size and the time required to generated this trace can be considerable.

For these reasons, we designed our emulator to support three execution modes: *full*, *incremental* and *replay*. In *full* emulation mode, the emulator executes the chain from scratch, starting from the provided entry point. The *replay* mode is completely based on an existing JSON trace and therefore it does not require any memory dump. This makes the rest of the analysis repeatable, and allows researchers to share traces without the need to transfer the content of the system memory (which may contain sensitive information and may be difficult to share for privacy reasons). Finally, the *incremental* mode is a combination of the previous two: it uses an input JSON trace (previously generated during a *full* emulation) and, once the last gadget in the trace is reached, it switches to *full* mode. This mode makes incremental analysis possible – a considerable advantage when dealing with very complex ROP chains.

Shadow Memory

The emulator initially reads the content of the memory from the memory dump. However, all write operations are redirected to a shadow memory area kept internally by the emulator. Subsequent read operations fetch data from the shadow memory (if the address has been previously written) or from the memory image.

Chain Boundary

Although the analyst knows the starting address of the first ROP chain, it is unclear where the chain ends. This problem is very important because we do not want to keep emulating instructions beyond the end of a chain, thus polluting the analysis with unrelated code.

Our framework solves the problem by using a number of heuristics. To start with, the emulator detects large increments or decrements of the *stack pointer*. Typically, during the execution of a single ROP chain, these deltas are small. Based on this locality principle, it is possible to find the exact moment in which the chain under analysis is terminated. This simple rule needed to be refined to take into account long jumps that may occur inside a single, very long chain (see, for instance, the case described in Section 6.5). By including heuristics based on the length of a gadget, and excluding the detected function invocations, our prototype was able to correctly stop the emulation process at the last gadget in our experiments. In case our heuristics fail, the analyst only needs to restart the emulator in *incremental mode* to continue the analysis of the chain from the point in which it was suspended.

Once the termination condition is triggered, the emulator stops and both the content of the shadow memory and the trace are saved to disk and are inspected to detect the presence of new ROP chains. If new stages are found, the emulator is re-started to analyze the next chain, and the process is repeated multiple times until all dynamically generated chains have been discovered and analyzed.

Syscalls and APIs

Complex ROP chains can invoke several system calls and library APIs, whose emulation is very complex (impossible in many cases) and goes beyond the scope of this paper. Our emulator recognizes when the execution is transferred to a system or API function, it saves its name in the trace, and then steps over its body to resume the emulation from the next gadget in the ROP chain.

This approach requires two types of information. First, the emulator needs to know the location and name of each API functions and system call routines. Luckily, this information can be easily retrieved by Volatility. Second, the emulator needs to know a valid output for each function. For instance, if the ROP chain allocates memory by calling `kmalloc`, the emulator needs to assign a valid (and not used) memory address to the function return value. Section 6.5 explains how

we handled, on a case-by-case basis, more complex cases that require complex buffers or data structures.

Multi-Path Exploration

In presence of long ROP chains with a complex control flow, a simple approach based on emulation is not enough to retrieve the entire ROP payload. The coverage is limited and takes into account only the executed branches – which often depend on the dummy return values generated by the emulator when the chain invokes system functions. This point is crucial for the analysis, as researchers need the entire chain to understand all the features and components of the ROP code. A simple emulation approach would likely miss important parts and thus some core functionalities may remain hidden.

We address this problem by introducing a multi-path emulation. Although this approach has its roots in the well-known multi-path exploration work proposed by Moser et al. [149], the original algorithm has been adapted to deal with ROP gadgets. In particular, our emulator is designed to recognize when the stack pointer is conditionally modified based on the content of the flag register. This pattern, however it is implemented, corresponds to a branch in the ROP chain. At the end of the emulation process, the JSON trace is analyzed to list all the branch points together with the value of the flags that was used in each of them by the emulator. The emulator is then re-started, this time providing an additional command-line parameter that specifies to complement the flag register at the required branch point, so that the execution can follow a different path. The exploration is terminated when all the branches have been explored. At the end, the analyst obtains several JSON traces containing different parts of the control flow graph.

However, in presence of loops in the ROP chain, the emulator could get trapped inside an endless execution path. The solution in this case is to keep track of the number of occurrences of the stack pointer during the execution of branch-related instructions. If this number is above a certain threshold (set to 10 in our experiments) the emulator automatically flips the flag bits to force the loop to end and explore the rest of the control flow graph.

6.4.3 Chain Splitting

The multi-path emulator generates a separate JSON trace file for each path in the ROP chain. The next step of our approach is in charge of splitting those traces, and removing duplicates parts that are in common between different traces. This part is divided in two passes. In the first step, every trace is cut at each branch point, and a new block is generated and saved in a separate JSON trace. During this operation the framework also records additional information describing the relationships among the different blocks.

Since conditional branch instructions are based on the value of the flag register, our system uses tests on the flags content or `pushf` operations as indicators of a

branch point. In particular, Chuck always pushes the flags on the stack to later retrieve them and isolate the `ZF` flag, whose values indicates which side of the branch needs to be taken. In the second pass, the chain splitter compares the individual blocks to detect overlapping footer instructions (i.e., gadgets in common at the end of different blocks) and isolate them in separate files.

The output of this phase is again a set of JSON trace file, this time not anymore associated to each individual path, but instead associated to each "basic block" in the chain. The chain splitter is implemented as a standalone python script.

6.4.4 Unchaining Phase

This phase transforms each JSON file into a sequence of instructions in the target architecture. This is obtained by applying a number of simple transformations. First, all the `ret`, `call`, and unconditional `jmp` instructions are removed from the trace. Then, `mov` instructions are simplified by computing their operands. In fact, due to the fact that immediate values are stored on the stack, ROP chains often contain expressions involving several registers (e.g., `mov rax, [rsp+0x30]`) that at this stage are replaced with their actual value. Similarly, we transform `pop` into `mov` instructions, by fetching the required values from the corresponding location on the stack.

6.4.5 Control Flow Recovery

The input of the control flow recovery is the set of `x86` binary blobs generated by the unchaining phase, plus some additional information specifying the way these blocks were connected in the traces generated by the emulator. The goal of this phase is to replace all the code that belongs to the gadgets used to implement ROP branches with more traditional and more compact conditional jumps.

This step is not trivial because it is necessary to switch from the stack pointer domain to the instruction pointer one. At every branch point, we need to create the instruction pointer logic from scratch to properly connect the two targets of a branch condition. In the case study, a simple conditional jump is implemented by 19 gadgets and 41 instructions. The proposed framework is able to recognize the condition and generate an equivalent assembly code in the instruction pointer domain. 19 gadgets are translated into two assembly instructions: a conditional jump – in our case represented by either `jz` or `jnz` instructions – and an unconditional jump (`jmp`).

The second task of the CFG recovery component is to detect and re-roll loops. ROP chains can contain both *return oriented loops* and *unrolled loops* that are programmatically generated when the chain is constructed. In the first case, the ROP instructions are used to conditionally repeat the same block of stack pointers, the same way a normal loop repeats the same sequence of EIP values. Unrolled loops repeat instead the same hard-coded sequence of gadgets over and over, for a pre-determined number of times.

For instance, Chuck uses unrolled loops to copy the dynamically generated chains to their final memory location. In fact, in the original source code of the rootkit (written in C), this is implemented as a short `FOR` loop that generates the appropriate gadgets. In the rootkit itself, it becomes a long sequence containing five gadgets repeated thousands of times. A simplified version of the gadgets looks like this:

```
pop rdx
mov [rax], rdx
pop rdi
add rdi, rax
mov rax, rdi
```

The value of the `rdx` register is taken from the stack, and then copied to a memory location pointed by the register `rax`. Finally `rax` is incremented by eight (the value of `rdi` taken from the stack).

Our tool is able to automatically identify these recurrent patterns and replace the entire sequence of instructions with a more compact snippet of assembly code representing a real loop with the same semantics. The resulting code is then wrapped withing a valid function prologue and epilogue and then embedded in a self-contained ELF file. It is important to note that it is not guaranteed that the file can actually be executed. If the original chain was part of a userspace shellcode, the ELF would probably contain all the instructions required to run the code. However, if the ROP chain is part of a kernel rootkit (as it is in our example), its code was initially designed to run in a very specific context in the kernel memory and therefore cannot be executed in a user-space program. However, our goal is just to generate a file that can be opened and analyzed by traditional reverse engineering tools such as IDA Pro.

6.4.6 Binary Optimization

The final step of our analysis consists of applying standard compiler transformations to optimize and simplify the generated code. For examples, dead code removal, simplifications of redundant mathematical operations, or global value numbering can greatly simplify the binary and makes it easier to understand for an analyst. However, these transformations have already been discussed in previous works [139, 203] and they are not the focus of our paper. Therefore, we implemented the most relevant and left further optimization to future work.

6.5 Evaluation

In this section we describe the experiments we conducted to evaluate ROPMEMU on the most complex ROP-based payload publicly available. All the ex-

Chain	Instructions	Gadgets	Blocks	Branches	Functions	Calls
Copy	414,275	184,126	1	-	-	-
Dispatcher	63,515	28,874	7	3	1	5
Payload	6320	2913	34	26	9	17

Table 6.1 – Statistics on the emulated ROP chains in terms of number of instructions, gadgets, basic blocks, branches, unique functions, and total number of invoked functions.

periments have been performed on an Ubuntu 14.04 `x86-64` running Python 2.7 and Volatility 2.4. The virtual machine containing the rootkit has been provided kindly by Chuck's author and runs Ubuntu Server 13.04 64-bit with UEFI BIOS.

6.5.1 Chains Extraction

In the first experiment we tested the ability of the multipath emulator of `ropmemu` to correctly extract the persistent chain (the *copy chain*), and the two dynamically generated chains (the *dispatcher chain* and the *payload chain*). The last two are volatile and they are only created in memory when the right conditions are triggered. The results are summarized in Table 6.1.

The emulator has been able to automatically retrieve the entire code of the three chains. The *copy chain* is the longest with 414,275 instructions, but it is composed of only a single basic block. The lack of a control flow logic makes this chain similar to a classic ROP shellcode, with the only difference of being composed of over 180K gadgets. This is a consequence of its main task: the creation and the copy in memory of the first dynamic component (*dispatcher chain*).

On the contrary, the *dispatcher chain* and *payload chain* have a lower number of gadgets and but they have a more complex control flow graph. In particular, the *dispatcher chain* has three branches and seven blocks. To recover the entire code, the emulator generated seven distinct JSON traces. The *payload chain* comprises instead 34 unique blocks and 26 branch points. This means the control flow graph has a more complex logic. It uses nine unique functions (`find_get_pid`, `kstrtou16`, `kfree`, `__memcpy`, `printk`, `strncmp`, `strrchr`, `sys_getdents`, and `sys_read` – the last two hooked by the rootkit) for a total of 17 function calls by exploring all the possible paths.

This experiment proves that `ropemu` can explore and dump complex ROP chains, impossible to analyze manually. We believe these chains show the limits of the current malware analysis to cope with return oriented programming payloads and the effectiveness of the proposed framework.

Chain	Initial State	Unchain Phase	CFG Recovery Phase
Copy	414,275	276,178	75
Dispatcher	63,515	40,499	16,332
Payload	6320	3331	2677

Table 6.2 – Number of instructions in each chain after each analysis phase

6.5.2 Transformations

In this experiment we show the effect of the other phases of our analysis on the extracted ROP chains. In particular, since it is impossible to show the entire code, we present the effect of the transformations on the payload size. The results are summarized on table 6.2. As shown in the third column, the *unchain* pass reduces considerably the ROP chain size (on average 39%). The *CFG recovery* pass filters out the instructions implementing the conditional statements, translates the chain from the stack pointer domain to the instruction pointer one, and finally applies the loop compression step. This transformations reduce the *copy chain* to only 75 instructions (starting from over 414K). The *payload chain* is less affected by these transformations because it contained ROP loops, but not any unrolled loop.

6.5.3 CFG Recovery

In the final experiment, we tested the ROPMEMU capability to retrieve and refine the control flow graph of a ROP chain as explained in section 6.4. Figure 6.2 and 6.3 illustrate the first phase on the *dispatcher* chain. In particular, Figure 6.2 represents the first version of the CFG, without any transformation. On Figure 6.3 we can observe the effects of the refinement steps. In these two figures every node represents a long stream of assembly instructions while the edges show the branch conditions.

The second step works on the binaries blobs and generates an ELF file. This ELF file connects all the blocks by leveraging the metadata information as explained in section 6.4 and can be inspected by ordinary reverse engineering tools. To test this functionality we opened the resulting file with IDA Pro. In Figure 6.4 we can observe the ELF representing the *copy chain* completely converted into the classic “EIP-based" programming paradigm. The graph is simple, there are no branches and the core functionalities are represented by the main loop highlighted in the picture. Figure 6.5 illustrates instead the *dispatcher chain* view on IDA Pro (for the sake of clarity every node is collapsed to have a cleaner view). The graph is similar to Figure 6.3, with just few additional nodes due to how the basic blocks are connected together. As expected, the shape of the graph is the same.

The control flow graph of the *payload chain* comprises 34 blocks and for space constraints its figure is not included in the paper.

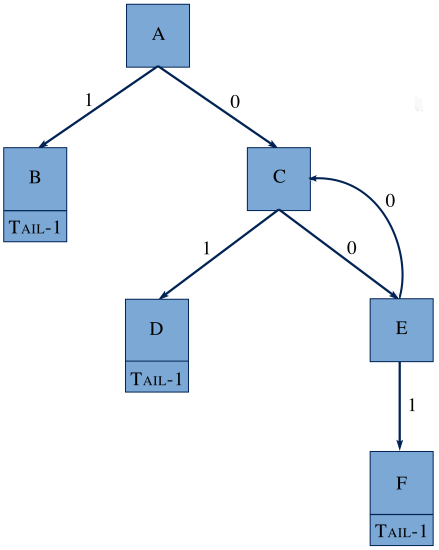

Figure 6.2 – Dispatcher - Raw CFG

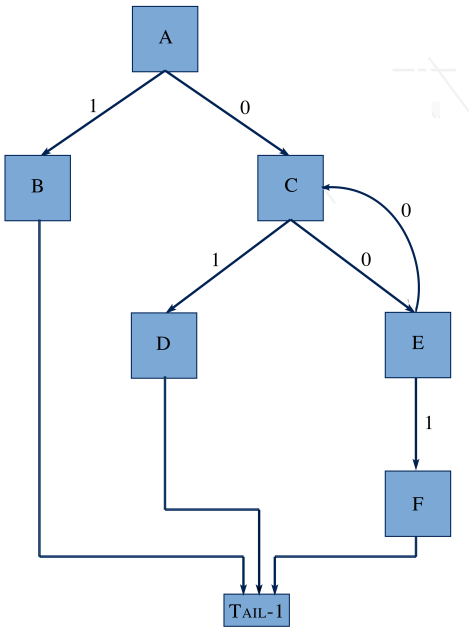

Figure 6.3 – Dispatcher - Final CFG

6.5.4 Results Assessment

An assessment system is fundamental to verify the results of the experiments. However, since it is not possible to run the final ELF to compare its behavior with the original rootkit, we needed to develop a number of dedicated verification tools.

We debugged the KVM virtual machine using GDB, that we extended with a set of Python plugins to extract information and compare them with the results of our framework. The assessment framework is working on the live virtual machine while ROPMEMU is working on a memory dump. The GDB plugins collect the state of the guest VM (memory and CPU) and the trace of all the executed instructions. These information are compared with the JSON traces generated by ROPMEMU.

We relied on this testing setup during development (to detect bugs in our code) and at the end of the experiments to verify that both the emulation of individual instructions and the entire lists of instructions in each ROP chain was correctly reconstructed by ROPMEMU. As a result, all the results presented in this paper match those found using the live GDB scripts.

6.5.5 Performance

The performance of our system largely depends on the emulation phase. The emulator is built on top of Volatility and the time required to perform the multipath emulation is linear in the number of instructions and the number of paths to emu-

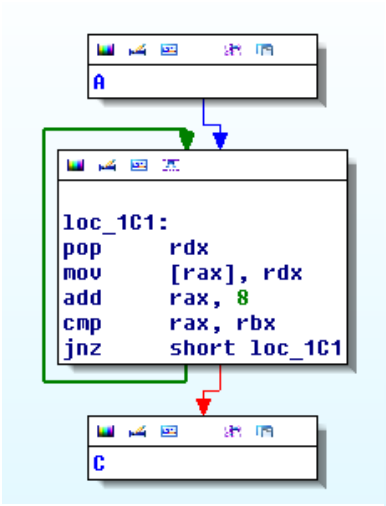

Figure 6.4 – Copy Chain - IDA Pro

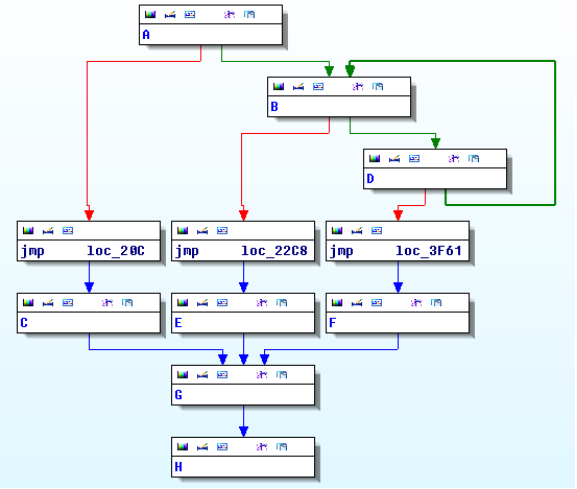

Figure 6.5 – Dispatcher Chain - IDA Pro

late. In average, our prototype emulates 133 instructions per second. For instance, the entire *copy* chain can be emulated in 52 minutes, while the *dispatcher* chain required 32 minutes to generate the three traces containing all the possible paths.

The performance of the *unchain* component depends instead on the size of the blocks to analyze. It ranges from the worst case of 61 minutes for *copy* chain (where everything is in a single huge block), to 3 minutes per block for the *dispatcher* chain (that is composed of smaller blocks). The *payload chain* traces have been generated on average in eight minutes while the *unchain* phase parsed each block in one minute. Overall, the entire analysis of the rootkit from the emulation to the final ELF binary took four hours.

All measurements have been recorded on a 16-Core Intel E5-2630 (2.3GHz) with 24GB RAM.

Chapter 7

Conclusions and Future Work

In this thesis, we presented a number of significant improvements to the current state of the art of modern malware and memory analysis. In the last years, these fields faced many challenges. Specifically, the increasing number of malicious samples forced the security community to devise more efficient and effective ways to automate the analyses. In particular, dynamic analysis with its most common deployed instance, *sandboxes*, redefined the whole industry. In the cases in which dynamic analysis shows its intrinsic limitations or is not enough for a real investigation, the sandboxes report may be complemented by memory analysis techniques. The combination of these two fields significanly eases the analyst tasks and is a step forward for both the industry and the academia.

The current state of the art addresses many problems of both fields. In particular, while malware and memory analyses have been improved from different angles, security companies still struggle to cope with the increasing number of malicious code. More importantly, these problems complicate the life of the final Internet users. We, as a community, need to investigate and research more in detail these critical topics in order to propose effective solutions able to eradicate the problem from the root. This dissertation is an attempt in this direction. The four contributions presented in this thesis advance the areas disccussed throughout this work: malware and memory analysis. Specifically, the first two aim at simplifying the analyst job and enhance key components of dynamic analysis. The last two shed light on advanced threats and propose memory forensics solutions to cope with these infections.

In chapter 3, we discussed the importance of looking at sample submissions from an intelligence and threat prevention point of view. We show that several binaries used in the most famous targeted attack campaigns had been submitted to our sandbox months before the attack was first reported. In this chapter, we propose a first attempt to mine the database of a popular sandbox, looking for signs of malware development. Our experiments show promising results: we were able to automatically identify thousands of developments, and to show how the

authors modify their programs to test their functionalities or to evade detections from known sandboxes.

Chapter 4 addresses the problem of network containment and repeatability in the context of dynamic analysis tools sandboxes. To tackle these problems, we described the implementation of Mozzie, a network containment system that can be easily adapted to all the existing sandbox environments. According to our experiments, an average of 14 network traces are required by Mozzie to model the traffic by approaching the problem of sandbox network emulation in a completely generic, protocol-agnostic way that can be applied to real-world malware samples. The benefits of the large-scale application of similar techniques are significant: old malware samples whose C&C infrastructure has been shut down can be analyzed in the same network conditions observed when they were active, in-depth analyses of samples of interest can be carried out in complete isolation, and malware targeting specific network environments (e.g. industrial control systems) can be analyzed in a replica of the expected network layout.

In chapter 5, we presented a first step toward the memory forensics analysis of hypervisors. In particular we discussed the design of a new forensic technique that starts from a physical memory image and is able to achieve three important goals: locate hypervisors in memory, analyze nested virtualization setups and show the relationships among different hypervisors running on the same machine, and provide a transparent mechanism to recognize and support the address space of the virtual machines. The solution we propose is integrated in the Volatility framework and allows forensics analysts to apply all the previous analysis tools to the virtual machine address space. Our experimental evaluation shows that Actaeon is able to achieve the aforementioned goals, allowing for a real-world deployment of hypervisor digital forensic analysis. In particular, the research presented in chapter 5 won the first Volatility plugin contest and has been integrated by Google in its Rekall suite [85].

Finally, chapter 6 represents the first attempt to automate the analysis of complex code implemented entirely using ROP. In particular, we discussed the challenges to reverse engineer programs implemented using return oriented programming and we proposed a comprehensive framework to dissect, recostruct and simplify ROP chains. Finally, we tested the framework with the most complex case proposed so far: a persistent ROP rootkit. The solution we described is ROPMEMU, and comprises a combination of Volatility plugins and additional standalone scripts. Our framework can extract the entire code of both persistent and dynamically generated ROP chains through a novel multipath emulation approach, simplify the output traces, extract the control flow graph and generate a final binary representing a cleaner version of the original ROP chain. The analysts can then operate on this binary with traditional reversing engineering tools like IDA Pro.

Although malware and memory analysis are two well studied fields and the contributions proposed in this thesis addressed some of their problems, there is still room for further improvements. Malicious programs afflict the life of users and the

business of companies. It is clear the solutions adopted so far are not effective. As a result, cybercriminals can compromise millions of machines, install implants and silently steal credentials. Unsurprisingly, the existing defense solutions prevent and detect only a small percentage of these ongoing attacks. I believe that in the future, the academic community has to work together with industrial partners. Only this synergy can make the Internet a safer place. Additionally, both sides have to raise awareness about the current cyber threats, their infection vectors and the importance of software updates. In this way, trivial infections such as phishing and breaches via exploit-kits may be contained.

Along the same line of this thesis, researchers should investigate more in detail the submissions received by online sandboxes. The work presented in Chapter 3 is meant to be just the first step in this direction. Security companies can use the collected information to detect new evasion techniques. It is also possible to directly link a new evasion technique to a malware family. In this way, researchers may fingerprint groups of malware authors from their peculiar probes. Regarding the containment phase, researchers should start sharing the finite state machines and create a common repository. Additionally, it is also necessary to publically release the source code of the protocol learning components and more research is needed to make these approaches more robust. In this way, external people can compare the different learning engines and adapt the core to their needs.

In the future, memory analysis researchers have to be ready to cope with advanced threats. In this direction, we proposed techniques to analyze hypervisors and return oriented programming codes. Similar approaches may be adapted for the analysis of other form of code reuse attacks such as sigreturn programming (SROP) and jump oriented programming (JOP). It is also necessary to have tools and techniques to acquire and analyze the physical memory segments in use by hardware devices such as graphics cards, network cards and the system BIOS. The protections in place on modern operating systems are forcing the attackers to install their stealthy components at lower levels. This is also confirmed by the recent Hacking Team documents leak with their UEFI persistent rootkit [96]. In addition, embedded systems are becoming more and more important in our lives, but the current memory analysis techniques are still at their early stages and do not fit well for the variaty of architectures and operating systems diversity. More in general, at the moment, the forensic community is not ready to face this variety of emerging threats and environments and this may compromise many ongoing and future investigations.

In conclusion, in this thesis we discussed four contributions to the fields of malware and memory analysis. The ideas presented in chapter 3 and chapter 4 can be adopted by security companies to improve their current systems. The contributions discussed in chapter 5 and chapter 6 reinforce the memory analysis fields. In particular, we added the support to cope with two advanced threats and provided forensic examiners with new techniques and a full-fledged system to use.

Chapter 8

Résumé

Au cours du temps, les cybercriminels ayant reconnu l'intérêt majeur de l'usage de l'Internet dans l'économie moderne, ont réalisé des activités lucratives au détriment des services en ligne. Pour conséquence, 317 millions de nouvelles variantes de logiciels malveillants ont été découverts en 2014. Ces logiciels malicieux ou "malware" ont causé des pertes financières significatives pour les entreprises et les particuliers. Une estimation des coûts est de l'ordre de 400 milliards de dollars pour l'année 2014. Les analyses modernes de logiciels malicieux sont très souvent effectuées de manière automatique; une minorité des malwares collectés sont analysés manuellement par des experts en rétro ingénierie. Ces logiciels malveillants sont surveillés dans des mécanismes de bacs à sable (Sandboxing). Ces bacs à sable sont des outils très pratiques et efficaces pour les analystes. Malheureusement ce procédé d'analyse offre bien souvent des limites dans l'étude de logiciel malveillant évolués. Pour cette raison, les analyses de logiciel malveillant sophistiqués sont souvent effectuées à partir d'éléments recueillis sur des ordinateurs infectés, au travers de capture ou "dump" mémoire. Cette thèse propose des améliorations pour l'analyse mémoire et l'étude des logiciels malveillants modernes. Bien que ces champs de recherches aient déjà été abordées au travers de perspectives différentes ces dernières années, il existe toujours plusieurs aspects qui peuvent être améliorés de manière significative. En particulier les bacs à sable ou 'Sandboxing' peuvent être plus efficaces en utilisant des techniques de contention réseau plus granulaires. Ainsi les chercheurs peuvent surveiller les soumissions de logiciel malveillants dans les systèmes de bacs à sable en ligne, et prioritiser les analyses manuelles. Dans le même esprit, l'analyse mémoire reste un domaine de recherche ouvert aux améliorations. Pour cela, nous proposons le premier framework logiciel permettant d'analyser des machines virtuelles et des hyperviseurs pour des configurations imbriquées. De plus, nous nous sommes appuyés sur des analyses mémoires pour faire face aux menaces avancées n'utilisant pas de technique d'injection de code.

8.1 Introduction

Il a été estimé que trois milliards de personnes ont été connectés à l'Internet en 2015 [6], et cela chiffre augmente chaque année que des régions entières dans les marchés émergents sont branché au cyberespace par les entreprises de télécommunication. L'Internet, et en particulier le World Wide Web (WWW), a simplifié la vie de millions de personnes et les entreprises. Aujourd'hui, de nombreuses familles ont une connexion Internet et propres plusieurs dispositifs tels que les ordinateurs, ordinateurs portables, et smartphones qui sont en mesure de se connecter au réseau. En conséquence, dans la dernière décennie de nombreuses activités d'affaires en ligne et même déplacés les gouvernements des institutions d'accueil de déplacer leurs services sur le Web.

Ce procédé offre plusieurs avantages aux utilisateurs finaux. Pour exemple, les gens peut acheter des produits en ligne et de faire des virements bancaires de leurs salons, webcams aident les gens à interagir avec leurs familles à l'étranger, et instantanée les programmes de messagerie permet une communication asynchrone libre. Dans générale, ces services réduisent les coûts et de gagner du temps pour les utilisateurs finaux.

Malheureusement, la révolution de l'Internet et de ses transformations ont aussi activités criminelles attirées. Miscreants réalisé l'activité lucrative derrière les services en ligne et a reconnu le rôle joué par Internet comme un pilier fondamental dans les économies modernes. En conséquence, plus de 317 millions de nouvelles variantes de logiciels malveillants ont été découverts dans 2,014 [189]. Dans la dernière décennie, les logiciels malveillants a été développé par groupes organisés pour le gain financier. Leur activité est basée sur le vol références et d'informations. Dans le cas où la machine ne contient pas des informations précieuses, il peut être loué à un tiers et utilisé pour envoyer du spam ou déni de service distribué (DDoS).

Plus récemment, les logiciels malveillants et les violations ont également été perpétrés par des puissants les gouvernements et les sociétés privées. Une guerre cachée est combattu avec les exploits et les rootkits afin d'exfiltrer information et à acquérir un avantage contre des adversaires. Dans ces cas, la ennemi peut varier de groupes de terroristes vers des pays légitimes, à partir de entreprises privées aux dissidents. Ces attaques sont silencieuses appelés menaces persistantes avancées (APT) et sont le composant central des campagnes de cyber-espionnage.

Le montant d'argent perdu par les citoyens et les entreprises privées en raison de cyberattaques ont atteint 400 milliards de dollars en 2014 [144]. Ceci est juste un rude estimation et ne prend pas en compte les dommages à la réputation, indirecte les coûts, les entreprises et compromis (comme Sony [194] Home Depot [197]) qui ne concernent pas divulguer publiquement leurs pertes financières. Par conséquent, les gens ordinaires et des entreprises privées dans les pays industrialisés exigent une protection pour leur comptes et la propriété intellectuelle. Les entreprises de sécurité jouent un rôle important et le rôle actif dans cette guerre sans fin. Ils offrent des solutions personnalisées à le secteur privé et public. Par exemple, les

ordinateurs flambant neufs sont expédiées avec un logiciel pré-installé antivirus, de nouvelles start-ups promettent d'éradiquer APT, les gouvernements forcé directives strictes ainsi que des certifications à garantir un niveau minimum de sécurité, les entreprises spécialisées vendent zero-day les exploits et les rootkits furtifs pour les interceptions légales, et les gouvernements paient formations avancées pour leur armée cyber.

Dans certaines opérations, les entreprises de sécurité ont uni leurs forces pour reprendre botnets et d'arrêter les auteurs de logiciels malveillants. En outre, ils ont créé d'immenses infrastructures pour collecter automatiquement et d'analyser le nombre croissant d'échantillons suspects. En fait, principalement en raison de l'emballage et le polymorphisme, les entreprises modernes anti-malware recueillir un nombre impressionnant de nouveaux échantillons par jour, par exemple une société connue comme Virustotal [198] recueille chaque jour plus d'un million d'échantillons.

8.1.1 Modern Malware Analysis

L'analyse des logiciels malveillants moderne est en grande partie automatisé, et seul un petit sous-ensemble des échantillons prélevés sont analysés manuellement par ingénierie inverse experts. Dans les dernières années, les entreprises de sécurité déployé un complexe infrastructures pour collecter des échantillons de leurs clients et de ad hoc machines vulnérables (honeypots). Ces infrastructures, souvent hébergés sur le nuage, analyser en temps réel le trafic des clients, et extraire des documents exécutables, et de les analyser à l'intérieur d'un environnement instrumentée (normalement appelé *sandbox*). Le système applique ensuite plusieurs heuristiques sur le rapports générés et une alarme est déclenchée dans le cas où le fichier est considéré comme mal intentionné.

Malheureusement, ce procédé présente plusieurs limites qui peuvent être exploitées par des logiciels malveillants de pointe. Par exemple, les échantillons peuvent être conçues pour détecter le environnement instrumenté et cacher leur comportement malveillant réel ou ils peuvent être programmé pour fonctionner uniquement sur une machine cible spécifique. Pour cette raison, l'analyse des logiciels malveillants sophistiqués implique souvent des informations d'exécution recueillies sur les systèmes infectés, généralement sous la forme d'un dépotoir de la mémoire physique. En fait, à partir de la mémoire de la machine infectée, il est possible d'extraire des objets importants et de recueillir des informations supplémentaires tandis que le malware fonctionne dans son environnement cible. La combinaison de ces deux approches est résumé dans la figure 8.1. L'analyste met à profit les deux approches, l'analyse dynamique binaire et analyse de la mémoire, à avoir une vue plus large sur une menace spécifique.

Cette thèse propose des améliorations au malware moderne et de la mémoire une analyse. Bien que ces domaines ont été étudiés à partir de différents points de vue dans les dernières années, il ya encore plusieurs aspects qui peut être considérablement améliorée. En particulier, les bacs à sable peuvent être optimisée de

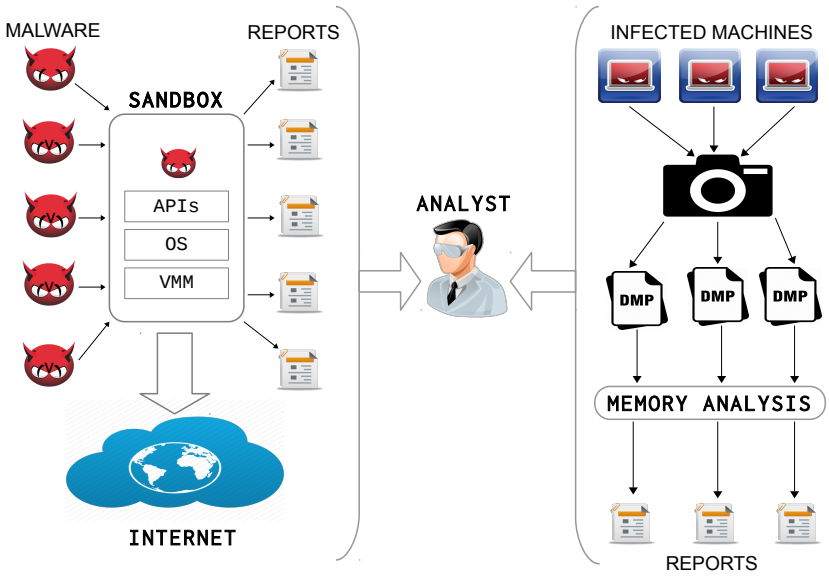

FIGURE 8.1 – Thesis overview

disposer de techniques de confinement de réseau plus granulaires. De plus, les chercheurs peuvent surveiller exécutables soumis à repérer des malwares actifs l'évolution de ces systèmes en ligne et de hiérarchiser les échantillons assigné pour une analyse manuelle.

Dans le même sens, l'analyse de la mémoire est encore un jeune champ avec une salle pour améliorations. Dans ce cas, nous avons proposé le premier cadre en mesure d'analyser les machines virtuelles et les hyperviseurs aussi quand configurations imbriquées sont en endroit. En outre, nous nous sommes appuyés analyse de la mémoire pour faire face aux menaces avancées qui ne nécessitent pas de techniques d'injection de code.

8.1.2 Sandboxing Technology

Solutions Sandbox sont un élément clé de l'analyse des logiciels malveillants moderne comme la nombre écrasant d'échantillons recueillis par jour fait d'autres solutions pas pratique. Par exemple, l'analyse manuelle ne échelle et exige des experts pour disséquer binaires malveillants. Deuxièmement, un système automatisé est nécessaire pour filtrer les échantillons non pertinentes et de recueillir de précieuses informations dans un laps de temps raisonnable. Troisièmement, les techniques d'analyse statiques sont souvent pas efficaces contre les fichiers malveillants. Pour aborder ces questions, les chercheurs en sécurité ont conçu un certain nombre de bac à sable environnements. Ces environnements instrumentés peuvent être exécutés en parallèle et peuvent être personnalisés par échantillon. Par

exemple, il est possible d'exécuter le même échantillon à la fois sur Windows XP et Windows 8. En outre, les analystes peut brancher des plugins supplémentaires pour étendre fonctionnalité de bac à sable (par exemple de simulation de l'activité de l'utilisateur).

Sandbox Design

Sandboxes sont conçus pour recueillir comportement des programmes malveillants. Malheureusement, échantillons de logiciels malveillants sont évasives et très complexe à analyser. Les chercheurs ont d'isoler et de surveiller activement les événements pertinents tels que le trafic réseau, Windows registre, des modifications du système de fichiers, la création de nouveaux processus, et opérations de mémoire suspicous. Afin de recueillir ces informations, les chercheurs peuvent décider de déployer un agent de surveillance invité en ce que recueille les informations en utilisant des techniques d'accrochage. Cet agent peut travailler à la fois comme un composante kernel- et espace utilisateur. La surveillance du noyau est nécessaire au moins en cas de noyau analyse de rootkit. Cette catégorie de logiciels malveillants travaille directement au ring 0 et altère la le noyau de système d'exploitation. Une autre approche consiste à déployer l'agent de moniteur out-of-invité. Dans ce cas, l'agent est mis en œuvre à l'intérieur de l'hyperviseur (ou émulateur) et la machine virtuelle introspection techniques sont utilisées. Malgré le fait que l'analyse dynamique des malwares a plusieurs avantages, dans la dernière année malware auteurs ont introduit une fonctionnalité anti-sandbox d'entraver l'analyse. Pour cette raison, les chercheurs en sécurité doivent soigneusement mettent en œuvre instrumenté environnements d'être aussi discret que possible. Sandboxes peuvent être construits sur le dessus de soit hyperviseurs ou émulateurs. Les deux approches ont avantages et les inconvénients et nous nous efforçons pour le même résultat.

Les solutions de virtualisation complète code instrument de l'hyperviseur. De cette façon, des chercheurs peuvent étendre les fonctionnalités du moniteur de machine virtuelle et introduire le nécessaire modules pour enregistrer l'activité des logiciels malveillants. Le composant du moniteur est en dehors du système d'exploitation invité, il devrait donc être impossible pour le malware à détecter le code d'instrumentation. En outre, la plupart des instructions sont exécutées directement sur le processeur, la surcharge supplémentaire est introduite seulement pour surveiller les événements d'intérêt particulier. Dans ces cas, les pièges de l'hyperviseur et exécute la routine propre à fiche information. Malheureusement, cette phase est pas anodin et il est compliquée par la soi-disant *semantic gap*. À ce stade, l'hyperviseur doit analyser la mémoire physique pour reconstruire le structures de données d'intérêt du système d'exploitation invité courante. C'est vraiment difficile et nécessite une profonde connaissance du système d'exploitation internes. Par exemple, les systèmes Windows ont trois points de vue différents sur le processus: `EPROCESS`, `KPROCESS` et `PEB`. Les deux premières structures de données garder une trace des informations vitales pour l'exécutif et les sous-systèmes du noyau

tandis que le dernier représente le processus en espace utilisateur. En outre, l'hyperviseur n'a pas de informations d'état de sorte que le mécanisme de distinguer les processus est en utilisant le `CR3` inscrire sur `x86` systèmes. Ce registre contient un physique l'adresse et les points directement à l'adresse de base de la première structure de données la mise en oeuvre de l'unité de gestion de mémoire (MMU). Tout événement est associé à un processus en inspectant ce inscrivez-vous et des informations supplémentaires sont récupérées grâce à des données correspondant structures (par exemple, `EPROCESS`). Cependant, il est pas trivial pour localiser et suivez ces structures de données. Le système doit heuristiques pour trouver le structures d'intérêt dans la mémoire physique et puis mettre en œuvre la traduction mécanisme (du virtuel au adresses physiques) à suivre indications intéressantes. Plus généralement, les chercheurs se réfèrent à cet ensemble de techniques que machine virtuelle introspection (VMI).

Communément, la virtualisation est utilisée comme technologie sous-jacente de démarrer exploitation guest operating systems Dans ces cas, le code de l'hyperviseur est pas instrumenté et le moniteur composant est implanté à l'intérieur de l'environnement virtuel. Dans ce scénario, ce composant peut être soit un pilote du noyau ou une bibliothèque dynamique (DLL) conçu se connecter fonctions d'intérêt ainsi que l'activité du réseau. Cette option nécessite des précautions particulières. Par défaut, les nouvelles machines virtuelles de marque installés sur haut de moniteurs les plus courants de la machine virtuelle, tels que VMWare, VirtualBox, Xen, KVM contient de nombreux témoignages à propos de l'environnement virtuel ainsi que sur le sous-jacent hyperviseur. Les chercheurs ont pour configurer ces machines et retirer trivial points de détection.

L'émulation est une technologie capable de simuler des instructions de montage via le logiciel. En conséquence, les émulateurs peuvent simuler des programmes complexes, tels que exploitation systèmes. Cette approche est flexible, l'émulateur peut mettre en œuvre ensemble des instructions de plusieurs architectures. De cette manière, il est possible d'observer programmes en cours d'exécution sur ARM sur le dessus de `x86` systèmes. Emulateurs fournissent environnements facile à utiliser instrumenté. En particulier, des solutions comme Qemu peuvent être prolongés par aux chercheurs d'enregistrer l'activité des malwares. Les approches d'émulation les plus courantes sont: émulation de système et le système complet émulation.

Émulation de système tente d'émuler via le logiciel le comportement de la système d'exploitation. Plus généralement, cette approche est en mesure de fournir le résultat de fonctions communes. Dans l'analyse des malwares et du système d'exploitation en place sur sandbox est Windows (allant de XP à la dernière version stable) - la système d'exploitation le plus touché par les logiciels malveillants. Ces OS communiqués sont généralement déployés avec service packs différents, certains logiciels malveillants peuvent montrer leur vraie nature que dans un environnement très spécifique. Dans cette configuration, les chercheurs doivent décider les fonctions d'intérêt. De manière générale, ils décident de surveiller suspecte appels, des fonctions comme `LoadLibrary`, `CreateRemoteThread`, `WriteProcessMemory`, etc sont correctement émulés. Sur les systèmes Win-

dows, il existe deux familles de fonctions: API Win32 et fonctions natives. Malheureusement, seulement les API Win32 sont bien documentés et considéré comme stable. Au contraire, les fonctions natives ont aucune documentation et peuvent être modifiées à n'importe quand. Les développeurs de Sandbox soutiennent API Win32 les plus courantes, mais ceux-ci fonctions sont un wrapper autour de natifs qui peuvent parler directement à la noyau. Les auteurs de malwares connaissent cette limitation et soustraire l'analyse basé sur une émulation de système en invoquant des API natives.

L'émulation du système complet est une technique capable d'émuler l'ensemble du fonctionnement système, cela est possible en soutenant périphériques matériels. Dans cette configuration, l'émulateur est en mesure de collecter toutes les instructions exécutée par le malware intérieur d'un système d'exploitation cible. Le composant du moniteur peut suivre toute la mémoire lecture et d'écriture. Toute cette information est vraiment utile au cours d'une analyse détaillée. En outre, le rapport a plus d'idées et, générale, il est plus facile pour un analyste de comprendre la nature de l'échantillon. Dans cette configuration, les instructions plus l'émulateur supporte, plus précise et furtive est l'analyse.

Émulation et de virtualisation approches sont flexible et facile à déployer. Les chercheurs en sécurité mis en place la machine, prendre un instantané et peut exécuter des milliers des échantillons sur le même hôte. Une fois que l'analyse est terminée, l'instantané est restaurée et le système est de nouveau propre. Malgré la limitation et l'possibile techniques d'évasion, ces deux solutions offrent un bon compromis. Dans des cas très spécifiques, l'analyste peut décider d'exécuter l'exemple sur un métal nu système. Le composant du moniteur peut être directement installé sur l'exploitation de l'hôte système. Dans ce cas, l'analyse est exacte, il n'y a pas de composants virtuels. Malheureusement, cette approche ne échelle. Une fois que la machine est infectée, le chercheur a à nouveau d'installer le système d'exploitation complet.

Problem Statement

L'analyse dynamique est une approche puissante pour découvrir comportement des programmes malveillants, et bacs à sable sont l'exemple le plus commun de cette technique. Ces instrumentées et environnements virtuels peuvent exécuter du code non sécurisé dans un isolé environnement et peut fournir à l'analyste une très souple et cadre d'analyse personnalisable. Sandbox Malheureusement, encore actuelles souffrent de plusieurs limitations. Dans cette thèse, nous nous concentrons sur deux problèmes dans le domaine de l'analyse dynamique des malwares.

Premièrement, l'analyse des logiciels malveillants est pas *repeatible*. En particulier, la comportement des programmes malveillants dépend souvent du contexte de réseau. Cela signifie que de nombreux échantillons d'interagir avec des serveurs en ligne et si ces serveurs ne sont pas disponibles le comportement (et donc le rapport d'analyse) est affectée. De plus, certains échantillons ne sont conçus pour fonctionner dans des environnements cibles spécifiques et échouerait lorsqu'il est

exécuté ailleurs. Malheureusement, la répétabilité est un très aspect important de l'analyse des logiciels malveillants et il est souhaitable dans de nombreux scénarios. Par exemple, les chercheurs peuvent vouloir analyser de nouveau le même échantillon après mois avec une nouvelle technique, afin de recueillir plus d'informations. Dans un infrastructures complètement automatisé basé sur des bacs à sable parallèles ce limitation peut entraver et de polluer l'analyse et les rapports.

Deuxièmement, l'emballage et le polymorphisme sont devenus très courants dans les logiciels malveillants et Aujourd'hui, il est courant d'avoir de nombreux échantillons différents pour la même famille. En conséquence, les bacs à sable sont surchargés par des binaires qui sont toutes équivalentes d'un point de vue du comportement. Ce phénomène complique la tâche de analystes de la sécurité. En particulier, la tâche de *distinctif* nouvelle et les logiciels malveillants importante du bruit de fond et polymorphe inintéressants échantillons est un problème ouvert très difficile dans le domaine.

8.1.3 Memory Analysis

L'analyse de la mémoire comprend un ensemble de techniques pour analyser le contenu des la mémoire de système (RAM). Dans la dernière décennie, il a gagné en popularité et il est maintenant une étape importante dans de nombreuses enquêtes réelles. Des chercheurs proposé techniques stables pour inspecter la mémoire physique, de localiser des données structures d'intérêt, et d'extraire les informations nécessaires. La popularité de cette approche réside dans le rôle central de la mémoire dans un système. En outre, les attaques avancées existent maintenant qui se trouvent uniquement dans la mémoire et ne laisser aucune empreinte à l'intérieur du système de fichiers.

L'analyse de la mémoire est un domaine de recherche actif qui a rapidement évolué au fil des ans. Elle peut être réalisée à la fois hors ligne (forensics mémoire) et dans un mode en ligne - mais les deux adopter des approches typiquement les mêmes techniques. Criminalistique mémoire est basées sur l'analyse des vidages de mémoire physiques, collectées par les outils et les dispositifs d'acquisition. En revanche, les systèmes d'analyse en ligne inspectent le système de mémoire vive. Ceci est possible en utilisant des programmes capables d'exporter un dispositif spécial qui permet un accès direct à la mémoire physique.

Les pratiquants doivent faire face à plusieurs défis. Le fossé sémantique est un problème commun. L'information est stockée dans la mémoire comme un flux brut d'octets et les experts ont besoin d'une profonde connaissance du système d'exploitation internes à extraire et reconstruire les artefacts nécessaires. Il ya plusieurs disponibles outils (comme la volatilité, Rekall, et de mémoriser) qui sont conçu pour faire face à ce problème. Ils commencent tous par la localisation importante structures de données. Ces structures de données peuvent résider dans la mémoire physique à un décalage fixe. Malheureusement, l'adoption croissante de techniques ASLR en espace noyau d'utilisateur et rend cette approche moins efficace contre le dernier OS de presse. Une approche plus fiable est basé sur la marche à travers

un nombre de structures de données intermédiaires (à partir de symboles globaux) dans afin d'atteindre les données cibles. Enfin, il est possible de créer une forte signatures pour balayer linéairement la mémoire physique et de découvrir tout occurrences d'un objet particulier. Cette phase est encore compliquée par la diversité du système d'exploitation, étant donné que la mise en mémoire des structures de données ne sont pas constante sur différents systèmes d'exploitation de presse. Par conséquent, les analystes besoin d'un profil dans lequel chaque structure de données est décrit en détail.

Un autre problème commun d'analyse de la mémoire est la traduction d'adresse. Une seul fois la phase de localisation est terminée, l'analyste a un objet contenant plusieurs domaines. Cependant, tout pointeur est une des adresses virtuelles et la mémoire cadre d'analyse fonctionne uniquement avec la mémoire physique. En particulier, la cadre doit mettre en œuvre sa propre unité de gestion mémoire (MMU). Ce implique la connaissance de l'architecture, généralement contenu dans le profil. Heureusement, les outils disponibles sont en mesure de répondre à toutes ces des défis.

Outre ces problèmes connus et faciles à résoudre, analyse de la mémoire, comme complémentaires approche, offre un unique, point de vue. Cette nouvelle perspective accélère considérablement l'analyse temps. Par exemple, les analystes peuvent immédiatement isoler les processus cachés. Plugins de volatilité comme `psxview` comparer la sortie de six différentes techniques pour la liste des processus en cours. De cette façon, les analystes peuvent facilement repérer les processus malveillants. En outre, dans le dernier années, l'analyse statique ont montré ses faiblesses. Plus précisément, les attaquants peuvent obscurcir leur code et entravent de manière significative l'analyse. La évolution de ces techniques a fait l'analyse statique presque inefficace. Par conséquent, les chercheurs ont adopté de nouvelles approches. En particulier, la mémoire criminalistique peuvent offrir de nouvelles perspectives pour l'analyste et, dans la plupart des les cas simples, est en mesure de vaincre les formes légères de l'obscurcissement telles que emballage. Un autre cas d'utilisation commune est l'analyse en profondeur d'une infection. Par exemple, les logiciels malveillants injecte communément code et DLL, même entières dans un autre espace d'adressage du processus, ceci est connu comme l'injection de code. Mémoire criminalistique propose des approches pour détecter automatiquement ces menaces (par exemple, `malfind` Volatility plugin).

Problem Statement

L'analyse de la mémoire est une approche complémentaire dans l'analyse de logiciels malveillants moderne. Ce est un domaine en croissance rapide qui a prouvé pour être utile dans de nombreux enquêtes - mais il a encore plusieurs limites. En particulier, dans cette thèse, nous explorons la façon dont les techniques d'analyse de la mémoire peut être étendue d'étudier deux formes de menaces avancées.

Tout d'abord, la médecine légale de mémoire est actuellement incapable de détecter et de faire face à toute forme de moniteur de machine virtuelle. Par consé-

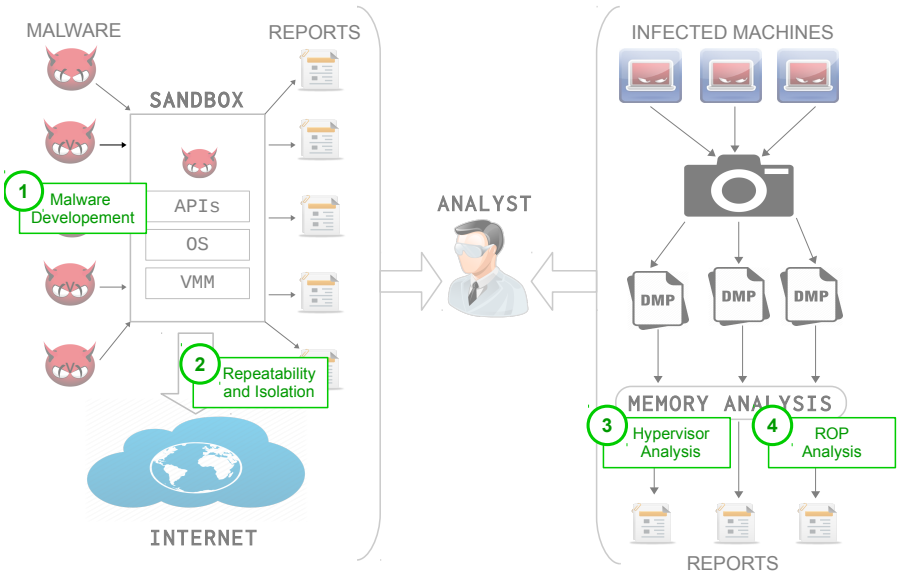

FIGURE 8.2 – Thesis Contributions

quent, tous les outils disponibles ne peut pas détecter et introspection transparente les systèmes d'exploitation invités. La situation est encore pire en présence des configurations imbriquées comme le malware analyste a pas d'outil pour détecter et de disséquer les hyperviseurs malveillants. En fait, sandbox ne prennent pas en charge la virtualisation imbriquée et, dans le meilleur de notre connaissance, il n'y a pas des outils et des techniques pour surveiller ces possibles menaces avancées.

Deuxièmement, les approches actuelles en médecine légale de mémoire visent à trouver des intrusions évidences dans les décharges de la mémoire physique. Généralement, ces preuves impliquent artefacts qui ont été créés ou injectés dans la mémoire par le malveillants Composants. Plugins de volatilité comme `psxview` et `malfind` sommes bon exemple d'outils qui effectuent cette tâche. Malheureusement, il est maintenant une tendance émergente des menaces avancées qui adoptent des techniques de réutilisation de code (tels que le rendement de la programmation orientée) en tant que moyen de brouillage, à effectuer calcul malveillants sans code injecté. Dans ces cas, à la fois outils de mémoire et d'analyse binaire sont totalement inefficaces pour localiser et disséquer cas de réutilisation de code, laissant ainsi l'analyste aveugles à cette nouveau type de menaces.

8.1.4 Contributions

Dans cette thèse, nous proposons un certain nombre de techniques permettant de résoudre non résolus problèmes dans les domaines de logiciels malveillants moderne et analyse de la mémoire. Dans notamment, la recherche présentée dans ce

document fait quatre individuelle Contributions: deux sur le domaine de l'analyse dynamique des malwares, et deux à améliorer la criminalistique de mémoire pour appuyer l'analyse des menaces avancées. Figure 8.2 montre les quatre contributions et comment ils sont situé par rapport à l'image générale.

Dans l'ensemble, nous avons fait les contributions suivantes:

1. Dans le chapitre 3 nous présentons une technique pour traiter des millions de demandes d'échantillons de malwares reçus par une analyse des programmes malveillants bac à sable et nous proposons une nouvelle méthodologie pour identifier automatiquement *développements* de logiciels malveillants. Notre approche est basée sur la combinaison d'analyse et de soumission de fichiers caractéristiques statiques et dynamiques. Nous aussi utiliser des techniques de data mining et d'apprentissage de la machine d'acquérir plus un aperçu sur la dynamique du développement des logiciels malveillants.
2. Dans le chapitre 4 nous étudions l'utilisation de l'apprentissage de protocole techniques pour modéliser le trafic généré pendant l'exécution de échantillons de logiciels malveillants pour répondre automatiquement les conversations de logiciels malveillants. En utilisant cette technique, nous avons développé un nouveau système de confinement de réseau et nous avons montré que, même avec certaines limites, il est possible de atteindre *confinement* complet et d'effectuer une *repeatible* analyse, également dans les cas où le comportement des programmes malveillants dépend hôtes externes comme C&C serveurs.
3. Dans le chapitre 5, nous avons proposé le cadre médico-légale première de mémoire à analyser les structures de l'hyperviseur de vidages de mémoire physique. De plus, nous avons analysé les configurations imbriquées et développé un mécanisme transparent de reconnaître et de soutenir l'espace d'adressage des machines virtuelles. Notre approche permet d'effectuer la criminalistique de mémoire pour analyser les hyperviseurs malveillants, comme ainsi que les machines virtuelles compromis qui font partie du grand virtualisé environnements.
4. Dans le chapitre 6 nous présentons un ensemble de techniques pour effectuer binaire et l'analyse de la mémoire des attaques sophistiquées qui ne reposent pas sur une injectée code. En particulier, notre approche identifie et discute de la principale défis qui compliquent code ingénierie inverse mis en œuvre à l'aide retourner programmation orientée (ROP). En outre, nous proposons une basée émulation cadre de disséquer, de reconstruire, et de simplifier ROP chaînes directement à partir d'un vidage de la mémoire physique. Nous avons testé notre outil avec l'exemple le plus complexe proposé à ce jour: un rootkit faite de plusieurs ROP chaînes, avec un total de 215,913 gadgets.

8.2 Related Works

Malware et l'analyse de la mémoire ont été largement étudiés dans la littérature. Malheureusement, l'industrie est encore aux prises avec de nombreux aspects de la analyse des logiciels malveillants. Plus précisément, les logiciels malveillants a évolué au fil des ans et nous avons assisté à son évolution d'un problème de niche à un fléau pour notre vie quotidienne. Dans cette guerre sans fin, les chercheurs tentent de chasser les auteurs de malwares et de protéger les utilisateurs.

Les premiers programmes malveillants étaient simples et conçus pour être un exercice technique. Teenegers intelligents et ennuyé codés programmes malveillants de montrer leurs compétences à la monde. Dans leurs créations n'y avait aucune intention de profit. Dans certains cas, la charge utile était un message texte. Dans d'autres cas, l'objectif était cybervandalism. Ther vecteurs d'infection étaient fondées sur des disquettes et que plus tard sur le réseau. En conséquence de ces premiers échantillons de virus, les premiers éditeurs de logiciels antivirus paru en 1987. Les premiers moteurs ont été basées sur le concept de *signature*. Immédiatement, les communautés de VX (groupes d'auteurs de virus) adapté leurs techniques et facilement contourné cette nouvelle contre-mesure. Il était 1989, lorsque le polymorphisme est apparu pour la première fois [34] et ce fut le début d'une course aux armements toujours en cours aujourd'hui.

La révolution de l'Internet a apporté de nombreux utilisateurs en ligne. Comme effet secondaire, beaucoup groupes de cyber-criminels sont apparus. Dans ces années, l'Internet a changé considérablement. Le cyberespace ne fut plus une place pour quelques personnes et intelligents et l'esprit underground disparu en raison de l'activité lucrative créée par les grandes sociétés. De l'autre côté, mécréants réalisé logiciels malveillants pourrait être utilisé comme un nouveau moyen de faire de l'argent. En conséquence, les entreprises de sécurité ont dû évoluer pour combattre organisée et des groupes qualifiés de cyber-experts.

Dans la dernière décennie, les chercheurs ont proposé plusieurs techniques pour faire l'analyse des programmes malveillants plus efficiente et efficace. Les premières approches ont été fondées sur une analyse manuelle et, par conséquent, de nombreuses avancées ont été proposées sur l'analyse statique et programme. Néanmoins, les auteurs de malwares conçus des formes avancées de dissimulation à entraver, l'analyse manuelle, et polymorphisme adoptée et le métamorphisme de contourner les approches de signature naïfs. Comme un Par conséquent, les entreprises de sécurité a investi des ressources sur l'analyse dynamique et, comme prévu, mécréants commencé à introduire des contre-mesures pour éviter le exectution sur les environnements instrumentés - dans un jeu permanent du chat et de la souris.

Le travail dans cette thèse couvre l'analyse des programmes malveillants modernes et automatisés. Cette approche est basée sur deux technologies: les bacs à sable et analyse de la mémoire. Dans ce chapitre, nous allons résumer les principales contributions à ces domaines. En particulier, dans la section 8.2.1 nous allons introduire les travaux connexes sur l'analyse des logiciels malveillants et techno-

logies de bac à sable. Dans la section 8.2.2 nous allons sommaire analyse de la mémoire.

8.2.1 Dynamic Malware Analysis

L'analyse dynamique des malwares exécute l'échantillon et observe son comportement au run-time. Tant l'exploitation forestière et le processus d'analyse peuvent être réalisés dans plusieurs façons et à différentes couches. En outre, l'environnement d'instrumentation dépend fortement sur le système d'exploitation en dessous. Pour ces raisons, au fil des ans, chercheurs ont proposé de nombreux environnements tirant parti des technologies différentes. Cette polyvalence et une salle pour d'autres personnalisations apportées sandbox le plus instance commune de l'analyse dynamique. À la fin années, les chercheurs ont amélioré de manière significative ces systèmes et, de nos jours, ils sont une composante importante activement utilisé par des entreprises de sécurité pour lutter contre les logiciels malveillants.

Cette technologie a évolué considérablement au fil des ans. Le premier rudimentaire approches consignés uniquement un sous-ensemble des événements d'intérêt. TTAnalyze [41] est le premier cadre global pour analyser des échantillons malveillants dans un environnement contrôlé environnement. L'évolution de ce projet est Anubis [19], le premier sandbox public en ligne. Aujourd'hui, il ya plusieurs bacs à sable dignes de mention. Certains d'entre eux sont disponibles gratuitement en ligne tels que Malwr [25], ThreatExpert [26] et Anubis [19]. D'autres sont ouverts source et peut être déployé en interne tels que Cuckoo [23] et Zerowine [107]. Autres bacs à sable (par exemple, Joebox [105], fireeye [80], Bromium [50] et Lastline [122]) sont propriétaires et un client peut avoir à la fois un accès en ligne et d'une instance privée. Toutes ces solutions permettent toujours rapport détaillé à l'analyste, mais la technologie sous-jacente et la mise en œuvre peuvent différer. Les premières versions de ces systèmes d'analyse de malwares pris en charge que les menaces de l'espace utilisateur et le moteur de l'exploitation forestière a été mis en œuvre à l'intérieur de l'invité système d'exploitation. En outre, la capacité d'enregistrement était tout simplement un système appels / API traceur. Certains sandboxes (par exemple CWSandbox [20] et Cuckoo [23]) utilisent leur propre bibliothèque d'accrochage tandis que d'autres préfèrent tirer parti des systèmes existants comme Detours [145]. Successivement, les experts en sécurité ont amélioré la composante de l'exploitation forestière à recueillir davantage d'informations et de montrer un rapport plus précis. Le support en mode noyau a été ajouté dans une deuxième phase. Bien que le nombre de ring0 échantillons est considérablement plus petit que le nombre de logiciels malveillants espace utilisateur, support du noyau est nécessaire d'avoir une idée immédiate sur le comportement du noyau complexe rootkits. Dans la troisième phase de plates-formes de sandbox, les chercheurs ont fait face à la intrumentation furtivité. La l'adoption généralisée de toutes les précautions spécifiques mentionnés ci-dessus a forcé les mécréants d'introduire routines anti-sandbox de l'échantillon logiciels

malveillants. De cette façon, les programmes malveillants ne divulguez pas leur comportement et passer inaperçu.

Ces fonctionnalités anti-sandbox sont conçus pour détecter l'environnement virtuel et la logiciel dessous (typiquement soit un hyperviseur ou un émulateur). Plus précisément, l'environnement virtuel peut contenir de nombreuses preuves. Par exemple, d'exploitation Windows systèmes sur le dessus de VirtualBox, un moniteur populaire de la machine virtuelle, peut être facilement détecté en regardant les appareils Windows Guest VirtualBox (`\\périphérique\\VBoxGuest`) ou les plages d'adresses MAC. En outre, le Registre de Windows est une autre source de preuves. Beaucoup de clés contiennent VirtualBox cordes. Bien que le nombre de ces contrôles possibles peut être sans fin, des chercheurs peut facilement patcher la grande majorité d'entre eux. Toutefois, les bacs à sable publics et en ligne doivent faire face avec d'autres points de détection simples. En particulier, l'environnement instrumenté a être randomisés contraire mécréants peuvent facilement détecter le bac à sable. Avtracker [3] montre ce problème et fournit des informations à facilement détecter les bacs à sable publics en ligne. L'auteur du site interagit régulièrement avec les services en ligne et accumuler des points possibles de détection tels que la IP publique, le nom d'utilisateur et l'ordinateur.

Les cybercriminels peuvent exploiter aussi d'autres points de détection. Ces points peuvent être plus problématique de patcher et de fixer et de résider dans les défauts du logiciel sous-jacent. Pire encore, un faible pourcentage est intrinsèque et montre la limite de la technologie utilisée (virtualisation ou l'émulation). Par exemple, le *attaques de synchronisation* exploiter ces intrinsèque limitations [39, 58, 68]. Dans cette situation, un auteur de logiciels malveillants peuvent exécuter la même instruction dans un environnement émulé et dans une machine physique. En conséquence, elle obtiendraient deux horodateurs différents. Après une phase de test, elle peut introduire la routine sur son malware contrôle et détecte l'écart de temps à l'aide `rdtsc` instructions de montage sur `x86` des machines. Cette astuce a été adopté par plusieurs familles de logiciels malveillants et il est souvent observé à l'état sauvage.

Le *bugs de l'émulateur* montrent les limites des approches de logiciels. Par exemple, les instructions Intel fixés est complexe et contient des milliers d'instructions. En conséquence, la la mise en oeuvre du logiciel de ces instructions peuvent contenir des bogues. En outre, le auteurs de l'émulateur peuvent décider de mettre en œuvre une partie seulement des les instructions et ignorer les effets secondaires rares. Par conséquent, dans certains cas, il est possible que l'exécution d'une instructions de montage sur un processeur virtuel peut se comporter différemment par rapport à un véritable et une cet écart peut être utilisé pour détecter l'environnement virtuel. En plus d'exploiter les bogues logiciels, les attaquants peuvent exploiter sans-papiers opcodes pour compliquer l'analyse. Par conséquent, le désassembleur de l'émulateur peut échouer à décoder les opcodes à une instruction de montage valide. Paleari et al. [157] ont élaboré un cadre automatisé pour détecter ces défauts. En particulier, ils ont étudié la mise en œuvre de la CPU dans Qemu et Bochs pour construire un ensemble de pilules rouges fiables. Les auteurs ont découvert 20,728

rouges pilules pour détecter Qemu et 2,973 pour la détection et Bochs. Dans cet ensemble de pilules rouges il ya aussi la pilule de l'origine Rutkowska [168]. Ce est important de noter que cette technique est générique et peut être appliquée à CPU virtualizer [143] ainsi que sur d'autres architectures. Encore une fois, ces tours ont déjà été observé à l'état sauvage.

Par conséquent, les chercheurs ont retiré les preuves les plus ordinaires de la virtuelle environnement et déplacé la technologie de l'exploitation forestière dans les hyperviseurs (ou émulateur) pour surmonter toute détection possible. Œuvres initiales, telles que celle proposée par Liston et al. [137], axées sur la suppression spécifique artefacts dans VMWare qui sont ciblés par des contrôles bien connus. Successivement, les praticiens ont déplacé les implants du système d'exploitation invité. Le premier travail dans cette direction et strictement liée à dynamique des malwares analyse est VMwatcher [102]. Ether [69] est le premier système d'analyse instrumentée transparente succès et théoriquement adressée tous les points de détection. Cependant, Pek et al. [161] trouvés et a réussi à détecter la environnement virtuel. Bien que l'éther et de systèmes similaires ont réussi à cacher leur présence, ils encourra inévitablement une pénalité de performance qui est prohibitif pour le déploiement sur les environnements automatisés d'analyse des logiciels malveillants réel à grande échelle. V2E [204] et DRAKVUF [129] visent à la transparence idéale et une performance optimale. Plus précisément, V2E combine transparence et d'efficacité pour une analyse en profondeur. L'outil comprend deux phases: enregistrer et rejouer. La première est basée sur KVM et est transparent tandis que le second est basé sur TEMU [46] pour d'autres inspections. DRAKVUF résout les défis techniques pour le soutien également out-of-the-box pour les rootkits noyau et met à profit les progrès dans le techology de virtualisation (par exemple, Extended Page Tables) d'avoir une faible surcharge d'analyse. Autres instrumentation cadres de l'ensemble du système et semblables à V2E et DRAKVUF sont construits sur des émulateurs et une commune choix est Qemu. Le premier cadre global est TEMU [46] de l'équipe BitBlaze et sa nouvelle et améliorée version DECAF [91]. S2E [59] fournit de puissants exécution symbolique fonctionnalité ainsi qu'une composante de traduire les Qemu IR (TCG) pour LLVM code binaire. Enfin, Panda [74] combine les caractéristiques de la précitée approches pour faciliter les tâches d'ingénierie inverse. En outre, il se concentre spécifiquement sur la répétabilité des analyses dynamiques et de la modularité du cadre, facilement étendre à travers plugins.

Bien que l'analyse dynamique est une arme puissante et un pilier dans l'analyse de logiciels malveillants moderne, il est pas parfait et peut être considérablement améliorée. Dans cette thèse, nous proposons deux avances à l'analyse dynamique des malwares. Le premier concerne le confinement de réseau afin d'obtenir une analyse reproductible. La seconde propose un ensemble de techniques pour surveiller les échantillons soumis à un sandbox de découvrir possibles développe-

Approach	Containment	Quality
Full Internet access	×	∼
Filter/redirect specific ports	∼	∼
Common service emulation	√	∼
Full isolation	√	×

TABLE 8.1 – Network access strategies in dynamic analysis

ments de logiciels malveillants. Pour cette raison, la partie restante de cette section se concentrer sur ces deux domaines.

Network Containment

Plusieurs stratégies ont été proposées pour résoudre le problème de containmnent du réseau et la qualité de la dynamique une analyse. En particulier, la notion de qualité se réfère à la fois à la nécessité de permettre la connectivité à des hôtes externes (pour exposer le comportement intéressant malware) et à la nécessité de rendre le processus d'analyse reproductible. Tableau 8.1 résume les travaux antérieurs dans quatre catégories différentes.

Full Internet access. L'approche la plus simple consiste à fournir la sandbox avec accès complet à Internet. Une approche similaire est toutefois inacceptable du point de vue de confinement: le malware est laissée libre pour propager aux victimes, ou de participer à d'autres types d'activités malveillantes (par exemple, DoS, spam). La qualité de l'analyse est aussi que partiellement acceptable: l'échantillon est laissé libre d'interagir avec des hôtes externes sur l'exécution, mais son comportement devient dépendante de l'état des hôtes externes, conduisant à la problèmes soulignés dans [125].

Filter/Redirect specific ports. Le problème de confinement associée à l'accès complet à Internet est rarement discutée dans des sandbox connectés à Internet tels comme Anubis [19], CWSandbox [20] et d'autres. De informelle discussions avec les responsables, il semble être une pratique courante pour le déploiement public de ces bacs à sable d'employer simple filtrage ou redirection règles, dans lequel les ports TCP communément associés aux analyses malveillants (par exemple le port 139 et le port 445) sont soit bloqué ou redirigé vers honeypots. Ce résout partiellement le problème de confinement: vulnérabilités de SMB sont très commune vecteur de propagation des logiciels malveillants d'auto-propagation, qui peut être facilement évitée avec de telles mesures. Cependant, cette approche ne sont pas en mesure de traiter avec d'autres types d'activités dont la nature ne peut pas être facilement discerné du TCP le port de destination. Une tentative similaire pour effectuer confinement par redirection également été mis en œuvre dans le cadre de honeyfarms tels que Potemkin [200] et GQ [64]. Dans de tels déploiements, la auteurs ont étudié l'idée de refléter le trafic

sortant généré par instances virtuelles infectés de la miellerie vers d'autres instances de la même miellerie. Une approche similaire avéré être utile pour l'analyse des programmes malveillants stratégies de propagation, mais n'a pas été efficace à traiter avec d'autres types de le trafic tel que C&C communication. En fait, redirigeant une tentative de connexion C&C miellerie générique vers une machine virtuelle est pas susceptible de générer des résultats significatifs. Kreibich et al. [117] ont récemment amélioré GQ qui en fait un réel et ferme malware polyvalent. Ils ont abordé le problème de confinement avec précision politiques, mais leur approche n'a pas abordé la question de répétabilité.

Common service emulation. Sandboxes tels que Norman Sandbox éviter le malware exécuté à partir de la connexion à Internet, et de fournir à la place implémentations de services génériques pour les protocoles courants tels que HTTP, FTP, SMTP, DNS et IRC. Une approche similaire a été revisité et amélioré par Ionue et al. dans [94], un deux-passer technique d'analyse des logiciels malveillants dans lequel l'échantillon malware est autorisé à interagir avec un "réseau miniature" générée par un émulateur en mesure de fournir une variété de mannequin Internet services à l'échantillon malware exécuté. Toutes ces approches sont toutefois Limited, et compter sur une connaissance a priori des protocoles de communication employé par le malware. Malware utilise souvent des variations de la norme protocoles ou complètement protocoles de communication ad hoc à-pas qui peut être traitées par les services factices. Yoshioka et al. [205]. Avoir tenté de résoudre ce problème par le raffinage de manière incrémentielle le confinement les règles en fonction des résultats d'analyse dynamique. Si une telle approche fourni une solution élégante au problème de confinement, il n'a pas abordé la qualité de l'analyse et il n'a pas tenté de supprimer les dépendances entre le comportement des programmes malveillants et l'état des hôtes Internet externes impliqués dans l'analyse.

Full Isolation. Empêcher complètement le malware d'interagir avec des hôtes Internent assure une parfaite maîtrise de son activité malveillante. Cependant, l'incapacité totale d'interagir avec C&C serveurs et les référentiels des composants supplémentaires est susceptible de fausser gravement les résultats de la dynamique processus d'analyse.

Tableau 8.1 souligne un compromis entre la partielle problème de confinement et de garantir la qualité et la répétabilité de la une analyse. D'une part, l'exécution du malware en pleine émulation traite de toutes les préoccupations, mais confinement, en interdisant l'échantillon de malware communiquer avec les hôtes externes dont il dépend, elle polarise fortement le résultats de l'analyse dynamique (par exemple, l'échantillon ne peuvent aller aussi loin que d'essayer pour se connecter aux hôtes mais sans exposer tout comportement malveillant réel). Sur d'autre part, fournir le bac à sable avec une connectivité Internet complète augmente la qualité

de l'analyse, mais il ne résout pas le problème de reproductibilité, et il soulève également des préoccupations éthiques et juridiques importants.

Dans le chapitre 4 nous abordons ce problème en explorant l'utilisation du protocole techniques pour créer automatiquement des modèles d'interaction de réseau pour l'apprentissage accueille le malware dépend lors de l'exécution (même en présence de la coutume et protocoles sans-papiers), et l'utilisation de ces modèles pour fournir le bac à sable avec un isolé mais riche environnement, de réseau.

Malware Development

Alors qu'il ya eu une grande quantité de recherche sur l'analyse des programmes malveillants et la détection, très peu de travaux dans la littérature ont étudié les ensembles de données recueillies par les logiciels malveillants sandbox d'analyse dynamique publics. Le plus étude approfondie dans ce sens a été menée par Bayer et al. [40]. Les auteurs ont examiné deux ans de Anubis [19] rapports et ils ont fourni des statistiques sur plusieurs l'évolution des logiciels malveillants et sur les types de comportements malveillants répandus observé dans leur ensemble de données.

Lindorfer et al. [135]. Mené la première étude dans le domaine de le développement de logiciels malveillants par l'étude de l'évolution dans le temps de onze connu familles de logiciels malveillants. En particulier, les auteurs ont documenté le malware processus de mise à jour et les changements dans le code pour un certain nombre de différent versions de chaque famille. Dans notre étude, nous observons le développement des logiciels malveillants processus sous un angle différent. Au lieu d'étudier les différentes versions de même les logiciels malveillants connus, dans le chapitre 3 nous proposons une détection à grande échelle de les auteurs du malware au moment où ils interagissent avec la sandbox lui-même.

Dans un papier différent, Lindorfer et al. [136]. Proposé une technique pour détecter les malwares sensibles environnement. L'idée est d'exécuter chacun plusieurs fois sur plusieurs échantillons de logiciels malveillants sandbox équipés différentes implémentations de surveillance et ensuite comparer les rapports normalisés de détecter les anomalies de comportement.

Étudie un domaine de recherche similaire, le phylogénie [89] des logiciels malveillants en utilisant des approches prises de le domaine de la biologie. Même si elle est partiellement liée à notre contribution, nous dans notre étude ne sont pas intéressés à comprendre la relation entre les différents espèces de malwares, mais seulement de détecter les soumissions suspectes qui pourraient être cadre d'une activité de développement de logiciels malveillants.

Dans un document de plus près à notre travail, Jang et al. [101] étudié comment déduire l'évolution du logiciel en regardant les binaires du programme. Dans notamment, les auteurs ont utilisé les deux fonctions d'analyse statiques et dynamiques pour récupérer la lignée de logiciels. Bien que le document de Jang a porté principalement sur bénigne programmes, certaines expériences ont également été menées sur 114 logiciels malveillants avec la lignée connue extraite de la Cyber Génome Projet [24]. Par rapport à notre travail, les auteurs ont utilisé un plus petit

un ensemble de caractéristiques statiques et dynamiques spécialement conçus pour déduire la lignée de logiciel (par exemple, le fait qu'un développement linéaire est caractérisé par une taille monotone croissante de fichier). Au lieu de cela, nous utilisons un ensemble plus riche de caractéristiques pour être en mesure de distinguer les développements de logiciels malveillants à partir des variations de les mêmes échantillons prélevés sur la nature et non soumis par l'auteur. Alors que nos approches partagent certaines similitudes, les objectifs sont clairement différent.

D'autres approches ont été proposées dans la littérature pour détecter similitudes entre les binaires. Flake [81] proposé une technique d'analyser les binaires sous forme de graphiques de graphiques, et nous avons été inspirés par son travailler pour le *d'analyse de flux de contrôle* décrit dans chapitre 3. Kruegel et al. [118] a proposé une technique similaire dans laquelle ils ont analysé les graphes de flux de commande d'un nombre de vers et ils ont utilisé une technique de coloration de graphe pour faire face à la problème graphique-isomorphisme.

Enfin, une étape de notre technique nécessaire pour regrouper similaire échantillons de logiciels malveillants. Il ya plusieurs articles dans le domaine de la classification des programmes malveillants [92, 97, 100, 201]. Cependant, leur but est de regrouper des échantillons appartenant à la même famille de logiciels malveillants aussi vite que possible et avec la plus grande précision. Ce est une tâche cruciale pour tous les éditeurs de logiciels antivirus. Cependant, notre objectif est différents que nous sommes intéressés à des échantillons de clustering basé uniquement sur binaire similitude et nous ne disposons pas de l'intérêt pour le clustering réuni des membres de la même famille sur la base de leur comportement.

8.2.2 Memory Analysis

Dans la dernière décennie, les investigations numériques ont considérablement évolué. Les chercheurs et les praticiens ont proposé efficiente et efficace méthodologies pour faire la criminalistique digitial scientifiquement comparable à la criminalistique traditionnels utilisés par les forces de l'ordre. Un rôle important dans cette évolution est représenté par la première Digitial Forensic atelier de recherche (DFRWS) en 2001 [158]. Merci à DFRWS, des universitaires et des experts en médecine légale ont uni leurs forces pour créer une communauté et systématiquement étudier le champ de proposer des outils et des méthodologies autant rigoureuse que possible. Au début, la criminalistique numérique assistés enquêtes de l'application des lois et, au fil des ans, les preuves recueillies ont été réglementées et accepté par le les tribunaux. En outre, la criminalistique numérique deviennent un domaine de recherche actif.

Criminalistique de mémoire est une branche de la criminalistique numérique et a été étudié intensivement depuis 2004, quand Carrier et al. [56] proposé Tribble, un dispositif d'acquisition de PCI à base de mémoire physique. En 2005, le DFRWS lancé une contester sur l'analyse de la mémoire. Comprend un vidage de la mémoire physique Le défi partir d'un Windows compromise systèmes et plusieurs questions au sujet de la violation d'exploitation. Répondre les questions, les cher-

cheurs ont dû créer de nouveaux outils et la technique pour analyser et extraire des informations du vidage de la mémoire. L'objectif des organisateurs était de motiver les chercheurs à étudier et d'améliorer cette recherche fascinante région. Dans la même année, et Movall al. [150] discuté une suite pour l'analyse de Linux physique mémoire. En 2006, Petroni et al. [162] présenté FATkit, un système modulaire cadre de l'inspection de la mémoire physique. FATKit supporte Linux et Systèmes d'exploitation Windows, la reconstruction de l'espace d'adressage (par exemple, IA-32) et a été développé suivant l'approche du transporteur sur l'abstraction couches [55]. L'évolution de FATkit est Volatilité [12], actuellement open source judiciaire de facto de mémoire cadre. Avant de volatilité et son FATkit prédécesseur, de nombreux chercheurs publié leurs propres outils personnalisés et des techniques pour extraire des objets simples (par exemple, la liste des processus). Tel est le cas de PTfinder de Schuster [175], la pmodump Stewart [186], les études de Kornblum [115, 116] et les publications de Dolan-Gavitt [72, 73], juste pour en nommer quelques-uns. Dans la même ligne, les chercheurs ont proposé outils de dumping [187,188] pour soulager la acquistion de la physique mémoire pour différents systèmes d'exploitation. En 2008, le DFRWS lancé un autre défi de l'analyse de la mémoire. Cette fois, la accent a été mis sur la création de méthodologies et d'outils pour l'exploitation Linux système [60]. Encore une fois, l'objectif des organisateurs était de favoriser les chercheurs et améliorer la champ. De même, en 2010 SSTIC contesté la communauté française de créer des outils pour l'analyse de la mémoire physique d'un dispositif en cours d'exécution Android [10,82]. Successivement, les chercheurs ont continué à améliorer le domaine médico-légal de la mémoire et a ajouté le support pour Mac OS X [12, 123] et FreeBSD [123].

En plus de solutions open source, de nombreuses entreprises ont créé leurs criminalistique de mémoire closed-source cadre. Ceci est le cas pour Mandiant (maintenant fireeye) avec Memoryze [141] et HBGary Responder Professional [90]. Ce phénomène montre l'intérêt du privé secteur sur la médecine légale de mémoire. Malheureusement, à l'heure actuelle, tous les cadres disponibles peut être facilement vaincu. Ceux-ci ont déjà été weakeness documentée par des universitaires [165] et indépendent chercheurs [98, 140, 180] mais les développeurs de la médecine légale de mémoire cadres ne traitent pas ces questions cruciales pour autant. Plus récemment, Case et al. [84] a analysé le cas nouveau les fichiers d'échange sur Linux et Mac OS X. comprimé RAM et largement étudiée De même, Cohen mis en œuvre et adapté le travail de Kornblum [116] dans Rekall [86], un spin-off de la volatilité proposé par Google, pour une analyse correcte et approfondie de le fichier d'échange Windows. En parallèle, les chercheurs ont testé et analysé la mémoire pour extraire beaucoup artefacts pas nécessairement liés aux composants du système d'exploitation (par exemple, les processus, les pilotes et les modules). Par exemple, Alex Halderman et al. [88], décrit plusieurs attaques où ils exploités DRAM effets de rémanence à récupérer clés cryptographiques et d'autres informations sensibles. Plus récemment, la soi-disant *attaque de démarrage à froid*

a été testé sur Android [151] et son efficacité a été confirmé tout cela n'a pas fonctionné comme prévu [87] sur des puces DDR3. En outre, la médecine légale de la mémoire ont été utilisées pour découvrir les programmes malveillants courir inaperçu sur l'ordinateur de la victime. Par exemple, Bianchi et al. [44] proposé *Blacksheep* d'identifier les machines infectées par un rootkit sur un infrastructure de cloud. Les auteurs ont construit une série de plugins de volatilité de comparer la des instantanés des machines différentes et mis en œuvre plusieurs heuristiques pour repérer évidences de rootkit. KOP [54] et MAS [66] appliquer des techniques d'analyse de la mémoire sur un seul machine pour localiser code malveillant courir au niveau du noyau, mais, malheureusement, ils ont besoin à la fois le code source du système d'exploitation. Plus récemment, MACE [79] étendu idée KOP mais en utilisant techniques d'apprentissage supervisé sur des pointeurs pour construire un noyau objets graphiques et détecter les rootkits noyau sans accès à la source code. Un autre avance intéressante a été présentée par Saltaformaggio et al. [173]. Avec DSCRETE, un système capable d'identifier l'information d'intérêt dans une décharge de la mémoire et restituer correctement son contenu à l'aide de sa propre logique de l'application. De cette manière, la analyste n'a pas besoin de connaître la configuration de la mémoire de structures de données contenant les renseignements qu'elle cherche.

2015 DFRWS défi axé nouveau sur la médecine légale de la mémoire, cette fois sur l'analyse des GPU mémoire [4] en fait, les chercheurs déjà proposées rootkits GPU à base [120] et observé dans les auteurs de logiciels malveillants sauvages mises à profit GPU à la mienne bitcoins [11]. Villani et al. [33] présenté un détaillée analyse des internes de GPU et décrit comment un médecins légistes peuvent faire face à ces des menaces. Dans la même ligne, à l'avenir, les analystes légistes ont à face menaces avancées et créer des outils et des techniques de disséquer et analyser ces nouvelles attaques. Dans cette thèse, nous allons améliorer le domaine en ajoutant le support pour localiser hyperviseurs (potentiellement malveillants) et les machines virtuelles sur les haldes de mémoire physique. Dans De plus, nous permettons à l'introspection déconnecté transparente de l'exploitation invité systèmes et de détecter les configurations imbriquées. Dans la littérature, les chercheurs ont hyperviseurs malveillants déjà proposés [70, 103, 111] que de un système d'exploitation hôte peut prendre le contrôle de l'ensemble de la machine. Plus récemment, ces menaces ont évolué et sont en mesure de porter atteinte à l'ordinateur ciblé directement à partir du BIOS [146]. Malheureusement, jusqu'à présent, aucune outils de médecine légale de la mémoire ont été en mesure de faire face à ces friandises. La deuxième contribution proposée dans cette thèse vise à détecter les attaques modernes et avancées qui ne injectez pas de code dans la victime système d'exploitation. Cette classe d'attaques est appelé *code reuse attacks* et ont de nombreux cas tels que ROP [178], JOP [48],BROP [47], SROP [49] et JIT-ROP [183]. Dans cette thèse, nous proposons une cadre fondé sur l'analyse de la mémoire et de l'émulation pour analyser et disséquer complexe Charges utiles ROP. Nous nous concentrons particulièrement sur ROP, car il est le plus commong par exemple observé dans la nature des attaques de réutilisation de code.

Hypervisors and Virtual Machines

Plusieurs documents ont proposé des systèmes à la recherche structures du noyau et de la mémoire utilisateur de l'espace dans la mémoire avec des méthodologies différentes. Dolan-Gavitt et al. [75]. Présenté un travail de recherche dans lequel ils ont généré automatiquement signatures robustes pour système d'exploitation importante structures. Ces signatures peuvent ensuite être utilisés par les outils d'analyse pour trouver la objets dans un vidage de la mémoire physique.

D'autres travaux ont porté sur la génération de fortes signatures pour les structures dans lequel il n'y a pas de valeurs invariant champs [130, 133]. Bien que ces approches sont plus générales et elles pourraient être utilisées pour notre algorithme, ils produisent un nombre important de faux positifs. L'approche que nous présentons dans chapitre 5 est plus ad-hoc, afin d'éviter les faux positifs.

Une autre approche générale a été présenté par Cozzie et al. dans leur système appelé Laika [62], un outil pour découvrir des structures de données inconnues en mémoire. Laika est basé sur des techniques probabilistes, en particulier sur apprentissage bayésien sans surveillance, et il a été prouvé être très efficace pour détection des malwares. Laika est intéressant car il est capable de déduire la bonne mise en page aussi pour les structures inconnues. Cependant, l'inconvénient est lié à l'exactitude et la quantité non négligeable de faux positifs et faux négatifs. Lin et al. ont développé DIMSUM [207] dans lequel, une donnée un ensemble de pages physiques et une définition de structure, de leur outil est capable de trouver les instances de structure, même si elles ont été unmapped.

Même si beaucoup de recherches ont été faites dans la médecine légale de mémoire terrain, au mieux de notre connaissance, il n'y a pas de travaux antérieurs sur criminalistique automatiques de virtualisation. Notre travail est le premier tenter de combler cette lacune.

Enfin, il est important de noter que plusieurs des auparavant systèmes présentés ont été mis en œuvre comme un plugin pour Volatility [13] - la norme de facto pour l'open criminalistique de mémoire source. En raison de l'importance de la volatilité, nous avons également décidé de mettre en œuvre nos techniques comme une série de différents plugins et comme un patch pour le noyau principal de ce cadre.

Advanced Threats

Return Oriented Programming (ROP) a été largement étudié dans le domaine scientifique la littérature à partir de plusieurs points de vue. Cependant, très peu de travaux ont nouvelles techniques présentées dédiés à l'analyse des chaînes ROP et dans cette section nous allons nous concentrer uniquement sur les recherches.

En ce sens, la première étude a été menée par Lu et al. [139]. Les auteurs proposés DeRop, un outil pour convertir ROP des charges utiles dans shellcodes normales, de sorte que leur analyse peut être effectuée par des outils d'analyse de

logiciels malveillants communs. Toutefois, les auteurs ont testé la efficacité de leur système que contre les exploits standard contenant chaînes ROP vraiment simple. Dans le chapitre 6, nous adoptons une partie de la transformations proposées par DeRop - que l'on se complètent par un certain nombre de nouvelles techniques nécessaires pour traiter avec les grandes et complexes chaînes d'un Rootkits ROP. Notre objectif principal est aussi plus ambitieux, que nous voulons parvenir à un couverture de code complet de la charge utile de ROP, également en présence d'dynamiquement chaînes générés.

Dans un autre document semblable à notre travail, Yadegari et al. [203] propose une approche générique de Code deobfuscate, dans lequel les auteurs considère ROP comme une forme de faux-fuyants. Leur système est basé sur l'analyse de souillure niveau bits que l'on applique aux traces d'exécution existants et peut être utilisé pour deobfuscate le graphe de flot de contrôle. En outre, le document adopte également des transformations similaires à la celles proposées par DeRop à manipuler des charges utiles ROP. Même si *Chuck* avait déjà été publié à l'époque, les auteurs ont fait valoir qu'aucun complexe par exemple des chaînes ROP était disponible, et ils ont testé le système contre les petits exemples avec une logique de flux de contrôle simple. En outre, le système proposé fait à ne pas émuler la chaîne de ROP et ne pas effectuer de couverture de code. Au lieu de cela, il met l'accent sur la simplification des traces d'exécution existants.

Une autre direction de recherche intéressante axée sur le problème de la localisation Chaînes de ROP en mémoire et potentiellement leur profil comportement [163, 185]. ROPMEMU peut tirer parti de ces techniques à identifier les chaînes ROP persistants. La phase de profilage proposé dans ces papiers étaient assez simple, et il peut échouer dans présence de chaînes de ROP complexes. Pour surmonter ces limitations, nous avons adopté une approche basée sur le processeur et l'émulation de la mémoire. Enfin, ces techniques ne fonctionnent pas en présence de chaînes de ROP emballés [138] ou des chaînes qui sont générées dynamiquement à runtime [199].

Jusqu'à aujourd'hui, tous les systèmes d'analyse et d'identification proposée dans le la littérature ont porté sur de simples exploits de l'espace utilisateur. Par conséquent, la technique présenté dans le chapitre 6 est la seule solution disponible qui supports l'analyse d'un véritable rootkit noyau mis en œuvre en ROP.

8.3 Conclusions and Future Work

Dans cette thèse, nous avons présenté un certain nombre d'améliorations significatives à l'état actuel de la l'art de logiciels malveillants moderne et analyse de la mémoire. Dans les dernières années, ces domaines confrontés à de nombreux défis. Plus précisément, le nombre croissant d'échantillons malveillants forcé la communauté de sécurité pour concevoir des moyens plus efficaces pour automatiser les analyses. En particulier, l'analyse dynamique avec son instance déployé le plus commun, *sandboxes*, redéfinir l'ensemble du secteur. Dans les cas dans les-

quels analyse dynamique montre ses limites intrinsèques ou ne suffit pas pour un vrai enquête, le rapport des bacs à sable peut être complétées par des techniques d'analyse de la mémoire. La combinaison de ces deux champs significanly facilite les tâches des analystes et constitue un pas en avant à la fois pour l'industrie et le milieu universitaire.

L'état actuel de l'art traite de nombreux problèmes de ces deux domaines. Dans particulier, alors que les logiciels malveillants et la mémoire des analyses ont été améliorées sous des angles différents, sociétés de sécurité ont encore du mal à faire face à l'augmentation du nombre de codes malveillants. Plus important, ces problèmes compliquent la vie des utilisateurs finaux d'Internet. Nous, en tant que communauté, besoin d'enquêter et de recherche plus en détail ces critiques sujets afin de proposer des solutions efficaces capables d'éradiquer le problème à la racine. Cette thèse est une tentative dans cette direction. Les quatre contributions présenté dans cette thèse avancer les zones disccussed long de ce travail: les logiciels malveillants et d'analyse de la mémoire. Deux Plus précisément, la première visent à simplifier le travail de l'analyste et de renforcer les éléments clés de l'analyse dynamique. Les deux derniers lumière de hangar sur les menaces avancées et proposer des solutions médico-légales de mémoire pour faire face à ces infections.

Dans le chapitre 3, nous avons discuté de l'importance d'examiner des échantillons soumis à partir d'un renseignement et le point de vue de la prévention de la menace. Nous montrons que plusieurs binaires utilisés dans le plus célèbre attaque ciblée campagnes avaient été soumis à nos mois de sandbox avant l'attaque était premier rapporté. Dans ce chapitre, nous proposons une première tentative pour exploiter la base de données d'un bac à sable populaire, à la recherche de signes de développement des logiciels malveillants. Notre expériences montrent des résultats prometteurs: nous avons pu identifier automatiquement des milliers de développements, et de montrer comment les auteurs modifient leur programmes de tester leurs fonctionnalités ou de se soustraire à des détections de connue sandboxes.

Chapitre 4 aborde le problème de confinement du réseau et répétabilité le contexte de la dynamique des outils d'analyse des sandboxes. Pour résoudre ces problèmes, nous avons décrit la mise en œuvre de Mozzie, un système de confinement de réseau qui peut être facilement adapté à tous les environnements de sandbox existants. Selon à nos expériences, une moyenne de 14 traces de réseau sont tenus par la Mozzie à modéliser le trafic en abordant le problème de l'émulation de réseau dans un bac à sable façon complètement générique, le protocole agnostique qui peut être appliqué à du monde réel échantillons de logiciels malveillants. Les avantages de l'application à grande échelle de techniques similaires sont importants: vieux échantillons de logiciels malveillants dont C&C infrastructures a été arrêté peuvent être analysés dans les mêmes conditions de réseau observées quand ils étaient actifs, des analyses approfondies d'échantillons d'intérêt peut être effectuée en complète isolement, et les environnements de réseau spécifiques de logiciels malveillants ciblant (par exemple, les systèmes de contrôle industriel) peuvent être analysés en une réplique de la configuration du réseau attendu.

Dans le chapitre 5, nous avons présenté une première étape vers l'analyse médico-légale de la mémoire de hyperviseurs. En particulier, nous avons discuté de la conception d'une nouvelle médecine légale technique qui commence à partir d'une image physique de la mémoire et est capable d'atteindre trois objectifs importants: localiser hyperviseurs dans la mémoire, analyser nichée configurations de virtualisation et de montrer les relations entre les différents hyperviseurs cours d'exécution sur la même machine, et de fournir une transparence mécanisme pour reconnaître et appuyer l'espace d'adressage des machines virtuelles. La solution que nous proposons est intégré dans le cadre de la volatilité et de permet criminalistique analystes d'appliquer tous les outils d'analyse précédents à la virtuelle espace d'adressage de la machine. Notre évaluation expérimentale montre que Actéon est en mesure d'atteindre les objectifs mentionnés ci-dessus, permettant une déploiement réel de l'analyse médico-légale numérique de l'hyperviseur. En particulier, la recherche présentée dans chapitre 5 a remporté le concours de plugin premier volatilité et a été intégré par Google dans sa suite Rekall [85].

Enfin, le chapitre 6 représente la première tentative d'automatiser l'analyse de code complexe entièrement mis en oeuvre en utilisant ROP. En particulier, nous avons discuté les défis pour inverser programmes de génie mis en œuvre en utilisant le retour programmation orientée et nous avons proposé un cadre global à disséquer, recostruct et simplifier les chaînes ROP. Enfin, nous avons testé la cadre avec le cas le plus complexe proposé à ce jour: un ROP persistante rootkit. La solution que nous avons décrit est ROPMEMU, et comprend une combinaison de Volatility plugins et des scripts autonomes supplémentaires. Notre cadre peut extraire la totalité du code des deux ROP persistante et généré dynamiquement chaînes à travers une approche d'émulation roman de trajets multiples, simplifient la sortie traces, extraire le graphe de flot de contrôle et génèrent un binaire final représentant une version propre de la chaîne de ROP originale. Les analystes peuvent alors fonctionner sur ce binaire avec des outils traditionnels d'ingénierie de recul comme IDA Pro.

Bien que les logiciels malveillants et d'analyse de la mémoire sont deux domaines bien étudiés et la contributions proposées dans cette thèse adressées certains de leurs problèmes, il ya encore de la place pour d'autres améliorations. Les programmes malveillants touchent la vie des utilisateurs et la affaires des entreprises. Il est clair que les solutions adoptées jusqu'à présent ne sont pas efficaces. En conséquence, les cybercriminels peuvent compromettre des millions de machines, installer et implants voler silencieusement pouvoirs. Sans surprise, les solutions de défense existants prévenir et à détecter seulement une petit pourcentage de ces attaques en cours. Je crois que, dans l'avenir, la communauté universitaire a à travailler ensemble avec des partenaires industriels. Seulement cette synergie peut faire d'Internet un endroit plus sûr. En outre, les deux parties doivent sensibiliser sur les cyber-menaces actuelles, de leurs vecteurs d'infection et l'importance

des mises à jour de logiciels. De cette façon, les infections banales telles que le phishing et les infractions via exploiter kits peuvent être contenues.

Le long de la même ligne de cette thèse, les chercheurs devraient étudier plus en détail les soumissions reçues par des bacs à sable en ligne. Le travail présenté dans le chapitre 3 est destiné à être juste la première étape dans cette direction. Les entreprises de sécurité peuvent utiliser les informations recueillies pour détecter les nouvelles techniques d'évasion. Il est également possible de relier directement une nouvelle technique d'évasion à une famille de logiciels malveillants. De cette façon, les chercheurs peuvent les empreintes digitales des groupes d'auteurs de logiciels malveillants à partir de leurs propres sondes. En ce qui concerne la phase de confinement, les chercheurs devraient commencer à partager le fini machines d'état et de créer un référentiel commun. En outre, il est également nécessaire pour libérer publiquement le code source de l'apprentissage de protocole composants et d'autres recherches sont nécessaires pour rendre ces approches plus robuste. De cette façon, les gens externes peuvent comparer les différents apprentissage moteurs et adapter le noyau à leurs besoins.

Dans l'avenir, les chercheurs de l'analyse de la mémoire doivent être prêts à faire face à menaces avancées. En ce sens, nous avons proposé des techniques pour analyser hyperviseurs et des codes de programmation de retour orientée. Des approches similaires peuvent être adapté pour l'analyse des formes d'attaques de réutilisation de code tels que sigreturn programmation (SROP) et saut programmation orientée (JOP). Il est également nécessaire de disposer d'outils et de techniques visant à acquérir et analyser les segments de mémoire physique utiliser par des périphériques matériels tels que les cartes graphiques, les cartes réseau et le système BIOS. La protections en place sur les systèmes d'exploitation modernes forcent les assaillants à installer leurs composants furtifs aux niveaux inférieurs. Ceci est également confirmé par le récent Hacking documents de l'équipe de fuite avec leur UEFI persistante rootkit [96]. En outre, les systèmes embarqués sont de plus en plus important nos vies, mais les techniques d'analyse de la mémoire en cours sont encore à leur début étapes et ne cadrent bien pour le targuer d'architectures et de fonctionnement la diversité des systèmes. Plus en général, à l'heure actuelle, la communauté légale est pas prêt à faire face à cette variété de menaces et environnements émergents et cela pourrait compromettre de nombreuses enquêtes en cours et futurs.

En conclusion, dans cette thèse, nous avons discuté quatre contributions aux domaines de programmes malveillants et analyse de la mémoire. Les idées présentées dans le chapitre 3 et chapitre 4 peut être adopté par la sécurité les entreprises à améliorer leurs systèmes actuels. Les contributions examinées au chapitre 5 et chapitre 6 renforcer les champs d'analyse de la mémoire. En particulier, nous avons ajouté le soutien pour faire face à deux menaces avancées et fourni les médecins légistes de nouvelles techniques et un système à part entière à utiliser.

Bibliography

[1] Amd's market share drops. http://www.cpu-wars.com/2012/11/amds-market-share-drops-below-17-due-to.html.

[2] Apple code signing. https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html.

[3] AV Tracker. http://avtracker.info/.

[4] Dfrws 2015 forensics challenge. http://www.dfrws.org/2015/challenge/index.shtml.

[5] Inception memory acquisition tool. http://www.breaknenter.org/projects/inception/.

[6] Internet live stats. http://www.internetlivestats.com/internet-users/.

[7] Microsoft Code Signing. https://msdn.microsoft.com/en-us/library/ms537361.aspx.

[8] Microsoft Driver Signing. https://msdn.microsoft.com/en-us/library/windows/hardware/ff544865\%28v=vs.85\%29.aspx.

[9] Nehalem architecture. http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf.

[10] Sstic 2010 challenge. http://communaute.sstic.org/ChallengeSSTIC2010.

[11] Trojan.badminer. https://www.symantec.com/security_response/writeup.jsp?docid=2011-081115-5847-99&tabid=2.

[12] Volatility framework: Volatile memory artifact extraction utility framework. http://www.volatilityfoundation.org/.

[13] Volatility framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility.

[14] Xtreme RAT. https://sites.google.com/site/xxtremerat/.

[15] Scapy. http://www.secdev.org/projects/scapy/, 2003.

[16] Bifrost Builder. http://www.megasecurity.org/trojans/b/bifrost/Bifrost2.0special.html, 2008.

[17] nfqueue-bindings. ://www.wzdftpd.net/redmine/projects/nfqueue-bindings/wiki/, 2008.

[18] Poison Ivy RAT. http://www.poisonivy-rat.com, 2008.

[19] Anubis. http://anubis.iseclab.org, 2009.

[20] Cwsandbox. http://www.mwanalysis.org, 2009.

[21] Netzob. http://www.netzob.org, 2009.

[22] A new approach to China. http://googleblog.blogspot.fr/2010/01/new-approach-to-china.html, 2010.

[23] Cuckoo Sandbox. http://www.cuckoosandbox.org, 2010.

[24] Darpa Cyber Genome Project. https://www.fbo.gov/index?s=opportunity\&mode=form\&id=c34caee99a41eb14d4ca81949d4f2fde, 2010.

[25] Malwr. https://malwr.com, 2010.

[26] ThreatExpert. http://www.threatexpert.com/, 2010.

[27] The Red October Campaign - An Advanced Cyber Espionage Network Targeting Diplomatic and Government Agencies. https://www.securelist.com/en/blog/785/, 2013.

[28] RDG Tejon Crypter. http://blackshop.freeforums.org/rdg-tejon-crypter-2014-t743.html, 2014.

[29] Tracking Malware with Import Hashing. https://www.mandiant.com/blog/tracking-malware-import-hashing/, 2014.

[30] VirusTotal += imphash. http://blog.virustotal.com/2014/02/virustotal-imphash.html, 2014.

[31] XtremeRAT: Nuisance or Threat? http://www.fireeye.com/blog/technical/2014/02/xtremerat-nuisance-or-threat.html, 2014.

[32] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[33] R. d. P. Antonio Villani, Davide Balzarotti. The Impact of GPU-Assisted Malware on Memory Forensics: A Case Study. August 2015.

[34] ArticleWorld. 1260. http://www.articleworld.org/index.php?title=1260_(computer_virus)&printable=yes.

[35] Aurelien Wailly. nROP. http://aurelien.wail.ly/nrop/.

[36] Axel “0vercl0k" Souchet. rp. `https://github.com/0vercl0k/rp`.

[37] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, NDSS 10, San Diego, CA, February 2010.

[38] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[39] E. Barbosa. Detecting virtualized hardware rootkits, 2007.

[40] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. In *USENIX workshop on large-scale exploits and emergent threats (LEET)*, LEET 09, April 2009.

[41] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, April 2006.

[42] F. Bellard. Qemu a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 2005.

[43] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: design and implementation of nested virtualization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[44] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[45] L. Bilge and T. Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 833–844, New York, NY, USA, 2012. ACM.

[46] BitBlaze Group. Temu. `http://bitblaze.cs.berkeley.edu/temu.html`.

[47] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.

[48] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, 2011.

[49] E. Bosman and H. Bos. We got signal. a return to portable exploits. In *Security & Privacy (Oakland)*, 2014.

[50] Bromium. Lava. `http://www.bromium.com/products/lava.html`.

[51] D. Bruschi, L. Martignoni, and M. Monga. Using Code Normalization for Fighting Self-Mutating Malware. In *Proceedings of the International Symposium of Secure Software Engineering (ISSSE)*. IEEE Computer Society, Mar. 2006. Arlington, VA, USA.

[52] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*, Oct. 2008.

[53] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *14th ACM conference on Computer and Communications Security*, pages 317–329. ACM New York, NY, USA, 2007.

[54] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 555–565, New York, NY, USA, 2009. ACM.

[55] B. Carrier. Defining digital forensic examination and analysis tools using abstraction layers. *International Journal of Digital Evidence*, 1:2003, 2002.

[56] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digit. Investig.*, 2004.

[57] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In A. Keromytis and V. Shmatikov, editors, *Proceedings of CCS 2010*, pages 559–72. ACM Press, Oct. 2010.

[58] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)*, pages 177–186, Anchorage, Alaska, USA, June 2008.

[59] V. Chipounov, V. Georgescu, C. Zamfir, and G. C. Selective symbolic execution. In *In Workshop on Hot Topics in Dependable Systems*, 2009.

[60] M. Cohen and D. Collett. Pyflag. `https://code.google.com/p/pyflag/`.

[61] Corelan. Mona. `https://github.com/corelan/mona`.

[62] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 255–266, Berkeley, CA, USA, 2008. USENIX Association.

[63] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *16th USENIX Security Symposium*, 2007.

[64] W. Cui, V. Paxson, and N. Weaver. GQ: Realizing a system to catch worms in a quarter million places. Technical report, ICSI Tech Report TR-06-004, September 2006.

[65] W. Cui, V. Paxson, N. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *The 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.

[66] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 601–615, Bellevue, WA, 2012. USENIX.

[67] A. Desnos, E. Filiol, and I. Lefou. Detecting (and creating !) a hvm rootkit (aka bluepill-like). *Journal in Computer Virology*, 7(1):23–49, 2011.

[68] A. Desnos, É. Filiol, and I. Lefou. Detecting (and creating!) a hvm rootkit (aka bluepill-like). *Journal in computer virology*, 7(1):23–49, 2011.

[69] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[70] Dino Dai Zovi. Hardware Virtualization Rootkits. `https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf`.

[71] B. Dixon. Watching attackers through virustotal. `http://blog.9bplus.com/watching-attackers-through-virustotal/`, 2014.

[72] B. Dolan-Gavitt. The VAD tree: A process-eye view of physical memory. *Digital Investigation*, 4, 2007.

[73] B. Dolan-Gavitt. Forensic analysis of the windows registry in memory. *Digital Investigation*, 5, 2008.

[74] B. Dolan-Gavitt, J. Hodosh, P. Hullin, T. Leek, and R. Whelan. Repeatable reverse engineering for the greater good with panda. In *Technical Report - Columbia University*, 2014.

[75] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 566–577, New York, NY, USA, 2009. ACM.

[76] T. Dumitras and D. Shou. Toward a standard benchmark for computer security research: The worldwide intelligence network environment (wine).

In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '11, 2011.

[77] S. Embleton, S. Sparks, and C. Zou. Smm rootkits: A new breed of os independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, 2008.

[78] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, pages 417–426, September 2010.

[79] Q. Feng, A. Prakash, H. Yin, and Z. Lin. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, 2014.

[80] FireEye. Fireeye as a service. `https://www.fireeye.com/products/fireeye-mssp-services.html`.

[81] H. Flake. Structural comparison of executable objects. In *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA*, pages 161–173, 2004.

[82] E. Girault. Volatilitux. `http://www.segmentationfault.fr/projets/volatilitux-physical-memory-analysis-linux-systems/`, 2010.

[83] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM.

[84] A. C. Golden G. Richard III. In Lieu of Swap: Analyzing Compressed RAM in Mac OS X and Linux. August 2014.

[85] Google - Rekall. Vm discovery and introspection with rekall. `http://www.rekall-forensic.com/posts/2014-10-03-vms.html`.

[86] Google - Rekall. Windows virtual address translation and the pagefile. `http://rekall-forensic.blogspot.ch/2014/10/windows-virtual-address-translation-and.html`.

[87] M. Gruhn and T. Muller. On the practicability of cold boot attacks. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 390–397. IEEE, 2013.

[88] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.

[89] M. Hayes, A. Walenstein, and A. Lakhotia. Evaluation of malware phylogeny modelling systems using automated variant generation, 2009.

[90] HBGary. Responder professional. `http://forensicswiki.org/wiki/HBGary_Responder_Professional`.

[91] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, 2014.

[92] X. Hu, S. Bhatkar, K. Griffin, and K. G. Shin. Mutantx-s: Scalable malware clustering based on static features. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 187–198, Berkeley, CA, USA, 2013. USENIX Association.

[93] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[94] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao. Malware behavior analysis in isolated miniature network for revealing malware's network activity. In *Proceedings of IEEE International Conference on Communications, ICC 2008, Beijing, China, 19-23 May 2008*, pages 1715–1721. IEEE, 2008.

[95] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Aug 2012.

[96] Intel Security. Hacking team's "bad bios": A commercial rootkit for uefi firmware? `http://www.intelsecurity.com/advanced-threat-research/blog.html`.

[97] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A static, packer-agnostic filter to detect similar malware samples. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 102–122, Berlin, Heidelberg, 2013. Springer-Verlag.

[98] Jake Williams and Alissa Torres. Add: Attention-deficit-disorder. `https://code.google.com/p/attention-deficit-disorder/`.

[99] James T. Bennett - FireEye. The Number of the Beast. `https://www.fireeye.com/blog/threat-research/2013/02/the-number-of-the-beast.html`.

[100] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 309–320, New York, NY, USA, 2011.

[101] J. Jang, M. Woo, and D. Brumley. Towards automatic software lineage inference. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 81–96, Berkeley, CA, USA, 2013. USENIX Association.

[102] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, 2007.

[103] Joanna Rutkowska. Bluepill. `http://web.archive.org/web/20080418123748/http://www.bluepillproject.org/`.

[104] Joe Damato. A closer look at a recent privilege escalation bug in Linux (CVE-2013-2094). `http://timetobleed.com/a-closer-look-at-a-recent-privilege-escalation-bug-in-linux-cve-`

[105] Joe Security LCC. Joebox. `http://www.joesecurity.org/joe-sandbox-technology`.

[106] Jonathan Salwan. ROPgadget - Gadgets finder and auto-roper. `http://shell-storm.org/project/ROPgadget/`.

[107] Joxean Koret. Zerowine. `http://zerowine.sourceforge.net/`.

[108] Kaspersky GReAT Team. Equation: The death star of malware galaxy. `http://securelist.com/blog/research/68750/equation-the-death-star-of-malware-galaxy/`, 2015.

[109] W. M. Khoo and P. Lio. Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families. *SysSec Workshop*, pages 3–10, 2011.

[110] S. T. King, P. M. Chen, Y. min Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *In IEEE Symposium on Security and Privacy*, pages 314–327, 2006.

[111] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[112] T. Koivunen. Sigbuster. `http://www.teamfurry.com"`, 2009.

[113] T. Kornau. Return oriented programming for the arm architecture. In *Master's Thesis - Ruhr-Universitat Bochum*, 2009.

[114] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, Supplement(0):91 – 97, 2006.

[115] J. D. Kornblum. Exploiting the rootkit paradox with windows memory analysis. *International Journal of Digital Evidence*, 5(1), Fall 2006.

[116] J. D. Kornblum. Using every part of the buffalo in windows memory analysis. *Digital Investigation*, 4(1):24–29, March 2007.

[117] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson. Gq: Practical containment for measuring modern malware systems. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, Berlin, Germany, November 2011.

[118] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 207–226, Berlin, Heidelberg, 2006. Springer-Verlag.

[119] G. Kurtz. Operation Aurora hit Google, Others. `http://web.archive.org/web/20100327181927/http://siblog.mcafee.com/cto/operation-\%E2\%80\%9Caurora\%E2\%80\%9D-hit-google-others`, 2010.

[120] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis. You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger. In *Proceedings of the 6th European Workshop on System Security*. EuroSec, Prague, Czech Republic, April 2013.

[121] N. Landwehr, M. Hall, and E. Frank. Logistic model trees. In *Machine Learning: ECML 2003*, pages 241–252. Springer Berlin Heidelberg, 2003.

[122] Lastline. Lastline. `https://www.lastline.com/platform/security-breach-detection`.

[123] K.-S. Lee. Volafox. `://github.com/n0fate/volafox`.

[124] C. Leita. *SGNET: automated protocol learning for the observation of malicious threats*. PhD thesis, University of Nice-Sophia Antipolis, December 2008.

[125] C. Leita, U. Bayer, and E. Kirda. Exploiting diverse observation perspectives to get insights on the malware landscape. In *DSN 2010, 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2010.

[126] C. Leita and M. Dacier. SGNET: a worldwide deployable framework to support the analysis of malware threat models. In *7th European Dependable Computing Conference (EDCC 2008)*, May 2008.

[127] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[128] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *21st Annual Computer Security Applications Conference*, December 2005.

[129] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[130] B. Liang, W. You, W. Shi, and Z. Liang. Detecting stealthy malware with inter-structure and imported signatures. In *Proceedings of the 6th ACM*

Symposium on Information, Computer and Communications Security, ASIACCS '11, pages 217–227, New York, NY, USA, 2011. ACM.

[131] M. Ligh. Using IDT for VMM Detection. `http://www.mnin.org/?page=vmmdetect`.

[132] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[133] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, 2011.

[134] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Atlanta, GA, USA, November 2008.

[135] M. Lindorfer, A. Di Federico, F. Maggi, P. Milani Comparetti, and S. Zanero. Lines of Malicious Code: Insights Into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[136] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting Environment-Sensitive Malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2011.

[137] T. Liston and E. Skoudis. On the cutting edge: Thwarting virtual machine detection. `http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf`, 2006.

[138] K. Lu, D. Zou, W. Wen, and D. Gao. Packed, printable, and polymorphic return-oriented programming. In *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011*.

[139] K. Lu, D. Zou, W. Wen, and D. Gao. derop: Removing return-oriented programming from malware. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, 2011.

[140] Luka Milkovic. Dementia. `https://code.google.com/p/dementia-forensics/`.

[141] Mandiant. Memoryze. `https://www.mandiant.com/resources/download/memoryze`.

[142] L. Martignoni, A. Fattori, R. Paleari, and L. Cavallaro. Live and Trustworthy Forensic Analysis of Commodity Production Systems. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2010.

[143] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi. Testing system virtual machines. In *Proceedings of the 19th international symposium on*

Software testing and analysis, ISSTA '10, pages 171–182, New York, NY, USA, 2010. ACM.

[144] McAfee. Net losses: Estimating the global cost of cybercrime. http://www.mcafee.com/mx/resources/reports/rp-economic-impact-cybercrime2.pdf.

[145] Microsoft. Detorus. http://research.microsoft.com/en-us/projects/detours/.

[146] Mikhail Utin. A myth or reality bios-based hypervisor threat. http://blog.deepsec.net/deepsec-2014-talk-a-myth-or-reality-bios-based-hypervisor-threat/.

[147] MORGAN MARQUIS-BOIRE, CLAUDIO GUARNIERI, AND RYAN GALLAGHER. Secret malware in european union attack linked to u.s. and british intelligence. https://firstlook.org/theintercept/2014/11/24/secret-regin-malware-belgacom-nsa-gchq/, 2014.

[148] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[149] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, 2007.

[150] P. Movall, W. Nelson, and S. Wetzstein. Linux physical memory analysis. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[151] T. Müller and M. Spreitzenbarth. Frost: Forensic recovery of scrambled telephones. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13.

[152] S. Needleman and C. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. J Mol Biol. 48(3):443-53, 1970.

[153] Nergal. Advanced return-into-lib(c) exploits. http://phrack.org/issues/58/4.html.

[154] Nguyen Anh Quynh. OptiROP: the art of hunting ROP gadgets. https://media.blackhat.com/us-13/US-13-Quynh-OptiROP-Hunting-for-ROP-Gadgets-in-Style-WP.pdf.

[155] Nicolas Economou - CoreSecurity. Agafi (Advanced Gadget Finder). http://www.coresecurity.com/corelabs-research/publications/agafi-advanced-gadget-finder.

[156] pakt. ropc. https://gdtr.wordpress.com/2013/12/13/ropc-turing-complete-rop-compiler-part-1/.

[157] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*, Montreal, Canada. ACM.

[158] G. Palmer. A road map for digital forensic research. In *Report From the First Digital Forensic Research Workshop (DFRWS)*, 2001.

[159] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*.

[160] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[161] G. Pék, B. Bencsáth, and L. Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, 2011.

[162] N. L. Petroni, J. Aaron, W. Timothy, F. William, and A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3, 2006.

[163] M. Polychronakis and A. D. Keromytis. Rop payload detection using speculative code execution. *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, 2011.

[164] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[165] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin. *Manipulating semantic values in kernel data structures: Attack assessments and implications*. 2013.

[166] D. Quist and V. Smith. Detecting the Presence of Virtualmachines Using the Local Data Table. http://www.offensivecomputing.net/files/active/0/vm.pdf.

[167] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi. When Hardware Meets Software: a Bulletproof Solution to Forensic Memory Acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, USA, December 2012.

[168] J. Rutkowaska. Red pill... or how to detect vmm using (almost) one cpu instruction. http://www.invisiblethings.org/papers/redpill.html, 2004.

[169] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. http://web.archive.org/web/20070911024318/http://invisiblethings.org/papers/redpill.html, 2004.

[170] J. Rutkowska. Subverting Vista Kernel for Fun and Profit. *Black Hat USA*, aug 2006.

[171] J. Rutkowska. Beyond The CPU: Defeating Hardware Based RAM acquisition. *Black Hat USA*, 2007.

[172] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, 2009.

[173] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu. Dscrete: Automatic rendering of forensic information from memory images via application logic reuse. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, Aug. 2014.

[174] sashs. ropper. `https://scoding.de/ropper/`.

[175] A. Schuster. Ptfinder. `http://computer.forensikblog.de/en/2007/11/ptfinder-version-0305.html`.

[176] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://users.suse.com/~krahmer/no-nx.pdf`.

[177] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[178] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, Oct. 2007.

[179] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, 2007.

[180] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for windows rootkit detection, 2005. `http://phrack.org/issues/63/8.html`.

[181] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[182] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[183] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, 2013.

[184] Solar Designer. Openwall. http://www.openwall.com/linux/README.shtml.

[185] B. Stancill, K. Z. Snow, N. Otterness, F. Monrose, L. Davi, and A.-R. Sadeghi. Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In *16th Research in Attacks, Intrusions and Defenses (RAID) Symposium*, Oct. 2013.

[186] J. Stewart. pmodump - truman project. http://www.secureworks.com/cyber-threat-intelligence/tools/truman/.

[187] M. Suiche. Moonsols windows memory toolkit. http://www.moonsols.com/windows-memory-toolkit.

[188] J. Sylve. Lime - linux memory extractor. https://github.com/504ensicslabs/lime.

[189] Symantec. Internet security threat report - april 2015. https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf.

[190] Symantec. The Stuxnet worm. http://go.symantec.com/stuxnet.

[191] Symantec. W32.Duqu, the precursor to the next Stuxnet. http://go.symantec.com/duqu.

[192] Symantec. W32.Koobface. http://www.symantec.com/security_response/writeup.jsp?docid=2008-080315-0217-99.

[193] Symantec Security Response. Regin: Top-tier espionage tool enables stealthy surveillance. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/regin-analysis.pdf, 2014.

[194] The Guardian. Sony hack: sacked employees could be to blame, researchers claim. http://www.theguardian.com/film/2014/dec/30/sony-hack-researchers-claim-sacked-employees-could-be-to-blame.

[195] The PaX Team. Pageexec. https://pax.grsecurity.net/docs/pageexec.txt.

[196] The PaX Team. Segmexec. https://pax.grsecurity.net/docs/segmexec.txt.

[197] The Wall Street Journal. Home depot hackers exposed 53 million email addresses. http://www.wsj.com/articles/home-depot-hackers-used-password-stolen-from-vendor-1415309282.

[198] VirusTotal - Google. Virustotal file statistics. https://www.virustotal.com/en/statistics/.

[199] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function hooks without code. In *Proceedings of the 21th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2014.

[200] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, 2005.

[201] G. Wicherski. pehash: A novel approach to fast malware clustering. In *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[202] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[203] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[204] Yan, Lok Kwong and Jayachandra, Manjukumar and Zhang, Mu and Yin, Heng. V2E: Combining Hardware Virtualization and Softwareemulation for Transparent and Extensible Malware Analysis. In *Proceedings of the 8th ACM SIGPLAN SIGOPS Conference on Virtual Execution Environments*, VEE12, New York, NY, USA, 2012. ACM.

[205] K. Yoshioka, T. Kasama, and T. Matsumoto. Sandbox analysis with controlled internet connection for observing temporal changes of malware behavior. In *2009 Joint Workshop on Information Security (JWIS 2009)*, 2009.

[206] K. Zetter. A google site meant to protect you is helping hackers attack you. `http://www.wired.com/2014/09/how-hackers-use-virustotal/`, 2014.

[207] L. Zhiqiang, R. Junghwan, W. Chao, Z. Xiangyu, and X. Dongyan. Discovering semantic data of interest from un-mappable memory with confidence. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS'12, 2012.

[208] D. A. D. Zovi. Hardware Virtualization Rootkits. *Black Hat USA*, aug 2006.