UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali

DOTTORATO DI RICERCA IN INFORMATICA
XXIII CICLO
SETTORE SCIENTIFICO DISCIPLINARE INF/01 INFORMATICA

Dealing with next-generation malware

Tesi di
Roberto Paleari

Relatore
Prof. D. Bruschi

Coordinatore del Dottorato
Prof. E. Damiani

Anno Accademico 2009/2010

UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE
XXIII CICLO

Dealing with next-generation malware

PhD Candidate
Roberto Paleari

Adviser
Prof. D. Bruschi

PhD Coordinator
Prof. E. Damiani

Academical Year 2009/2010

Abstract of the dissertation

Dealing with next-generation malware

by
Roberto Paleari

Doctor of Philosophy
in
Computer Science

Università degli Studi di Milano
2011

Malicious programs are a serious problem that threatens the security of billions of Internet users. Today's malware authors are motivated by the easy financial gain they can obtain by selling on the underground market the information stolen from the infected hosts. To maximize their profit, miscreants continuously improve their creations to make them more and more resilient against anti-malware solutions. This increasing sophistication in malicious code led to *next-generation malware*, a new class of threats that exploit the limitations of state-of-the-art anti-malware products to bypass security protections and eventually evade detection. Unfortunately, current anti-malware technologies are inadequate to face next-generation malware. For this reason, in this dissertation we propose novel techniques to address the shortcomings of defensive technologies and to enhance current state-of-the-art security solutions.

Dynamic behavior-based analysis is a very promising approach to automatically understand the behaviors a malicious program may exhibit at run-time. However, behavior-based solutions still present several limitations. First of all, these techniques may give incomplete results because the execution environments in which they are applied are synthetic and do not faithfully resemble the environments of end-users, the intended targets of the malicious activities. To overcome this problem, we present a new framework for improving behavior-based analysis of suspicious programs, that allows an end-user to delegate security labs the

execution and the analysis of a program and to force the program to behave as if it were executed directly in the environment of the former. Our evaluation demonstrated that the proposed framework allows security labs to improve the completeness of the analysis, by analyzing a piece of malware on behalf of multiple end-users simultaneously, while performing a fine-grained analysis of the behavior of the program with no computational cost for the end-users.

Another drawback of state-of-the-art defensive solutions is non-transparency: malicious programs are often able to determine that their execution is being monitored, and thus they can tamper with the analysis to avoid detection, or simply behave innocuously to mislead the anti-malware tool. At this aim, we propose a generic framework to perform complex dynamic system-level analyses of deployed production systems. By leveraging hardware support for virtualization available nowadays on all commodity machines, our framework is completely transparent to the system under analysis and it guarantees isolation of the analysis tools running on top of it. The internals of the kernel of the running system need not to be modified and the whole platform runs unaware of the framework. Once the framework has been installed, even kernel-level malware cannot detect it or affect its execution. This is accomplished by installing a minimalistic virtual machine monitor and migrating the system, as it runs, into a virtual machine. To demonstrate the potentials of our framework we developed an interactive kernel debugger, named HyperDbg. As HyperDbg can be used to monitor any critical system component, it is suitable to analyze even malicious programs that include kernel-level modules.

Despite all the progress anti-malware technologies can make, perfect malware detection remains an undecidable problem. When it is not possible to prevent a malicious threat from infecting a system, post-infection remediation remains the only viable possibility. However, if the machine has already been compromised, the execution of the remediation tool could be tampered by the malware that is running on the system. To address this problem we present Conqueror, a software-based attestation scheme for tamper-proof code execution on untrusted legacy systems. Besides providing load-time attestation of a piece of code, Conqueror also ensures run-time integrity. Conqueror constitutes a valid alternative to trusted computing platforms, for systems lacking specialized hardware for attestation. We implemented a prototype, specific for the Intel x86 architecture, and evaluated the proposed scheme. Our evaluation showed that, compared to competitors, Conqueror is resistant to both static and dynamic attacks.

We believe Conqueror and our transparent dynamic analysis framework constitute important building blocks for creating new security applications. To demonstrate this claim, we leverage the aforementioned solutions to realize HyperSleuth, an infrastructure to securely perform live forensic analysis of potentially compromised production systems. HyperSleuth provides a trusted execution environment that guarantees an attacker controlling the system cannot interfere with the analysis and cannot tamper with the results. The framework can be installed as the system runs, without a reboot and without loosing any volatile data. Moreover,

the analysis can be periodically and safely interrupted to resume normal execution of the system. On top of HyperSleuth we implemented three forensic analysis tools: a lazy physical memory dumper, a lie detector, and a system call tracer. The experimental evaluation we conducted demonstrated that even time consuming analyses, such as the dump of the content of the physical memory, can be securely performed without interrupting the services offered by the system.

To Laura,
Thank you for always believing in me

Acknowledgements

This dissertation would have not been possible without the help of several people, who supported me during my years as a PhD student.

First, I would like to thank my advisor, Prof. Danilo Bruschi, for his guidance and for having always encouraged me and supported my research ideas. I am also extremely grateful to my external referees, Prof. Herbert Bos, Prof. Wenke Lee, and Prof. Dawn Song: their insightful comments and suggestions greatly contributed to improving by dissertation. I would like to thank them all so much for the time they spent on my thesis.

There are many friends I would like to thank. First of all, I wish to thank Lorenzo Martignoni: he is really one of the best computer scientists I have had the fortune to meet, besides being a great friend; I really learnt a lot by working with him, and I known we will still have a lot of fun together. I am also very grateful to Lorenzo "Sullivan" Cavallaro and Andrea Lanzi: I thank them for their help and for the great discussions we had together; I always admired their tenacity, and I have been very fortunate to collaborate with them. A special thank goes to Prof. Mattia Monga: his knowledge and his love for research have inspired me during my doctoral studies.

I want to thank all those who shared with me their PhD experiences, especially Emanuele Passerini – I will always remember the night we spent jailbreaking our iPods –, Giampaolo Fresi Roglia – his final solution to the DefCon's Shakespeare challenge is a masterpiece –, and Alessandro Rozza: we really had a lot of fun together! Of course I am also indebted to the LaSeR crew, namely Aristide Fattori, Luca Giancane, Davide Marrone, Alessandro Reina, and all the students who spent some time in the lab. I will never forget the days – and nights – we spent working on crazy (and extremely "low-level") projects, or just participating in our beloved CTF competitions. Guard@MyLan0 (a.k.a. Chocolate Makers) team rocks! I hope I will have the opportunity to work with them again in the future.

To conclude, I wish to extend a huge thank you to my parents for their help.

They always supported all the decisions I made, and they never, ever asked me for anything in return. I know it is not so easy to tolerate a moody PhD student, so I need to thank them both very much!

Last but not least I would like to thank my dear Laura. She has always been (and always will be) a big source of inspiration for me: without her I would simply not be the person I am today. This dissertation is dedicated to her.

Contents

1 Introduction

The term *malware*, or malicious software, indicates any computer program written with the explicit intent to damage users and compromise their systems to perform various types of fraud. Nowadays, malware is used to send spam e-mails, to perpetrate web frauds, to steal personal information, and for many other nefarious tasks. The widespread diffusion of malicious software is one of the biggest problems the Internet community has to face today.

Malicious programs have been known for decades, since in 1982 the Elk Cloner virus started to spread by infecting floppy disks. Despite this first incident, the malware phenomenon was able to gain significant media attention only in 1988, when the Internet Worm infected most of the Internet at the time [121]. The community soon realized this was not an isolated case, and the next events firmly confirmed this belief. Indeed, the problem quickly became so extensive that it hit the headlines many times [38, 70], and in the following years malware caused an economic damage of many billions of dollars [28, 77].

To defend against malicious programs, users typically rely on anti-malware products in order to preemptively detect a threat before it can damage their systems. The approach adopted by traditional anti-malware solutions is *signature-based*: malware samples are analyzed in a security laboratory and, for each sample, analysts identify a signature, i.e., a sequence of bytes that uniquely identifies the sample and that is unlikely to be found in benign programs. Then, anti-malware products are shipped with a database of signatures. If a suspicious application is found to contain a known signature, the application is considered to be infected by the corresponding malicious sample. To account for newly discovered samples, vendors periodically distribute signature updates to all their customers [129].

For some years, the signature-based approach has been quite effective, but more recently we have witnessed a drastic shift in the malware landscape. Criminals realized that malicious programs can turn into a very profitable business, and started to use malware to perform all kinds of illegal activities. As depicted in Figure 1.1, this trend is also testified by the exponential growth in the num-

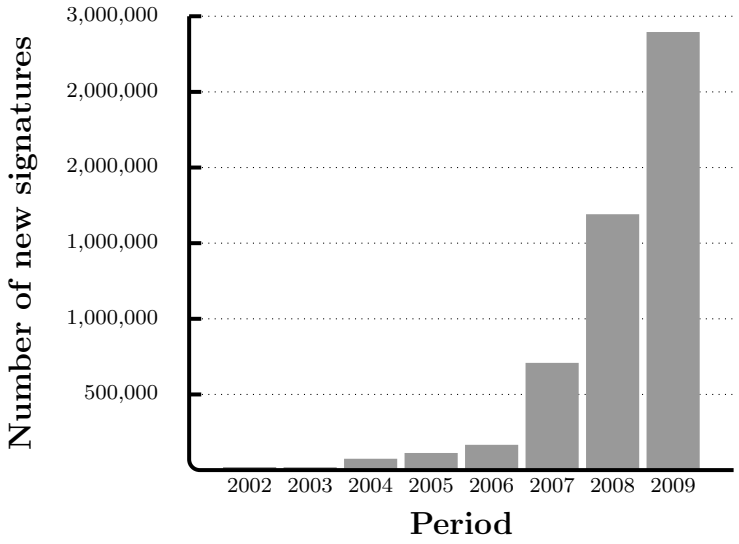

Figure 1.1: New malicious code signatures observed in the last years (*source*: Symantec Corporation [126])

ber of malicious threats observed during the last years. However, the problem goes far beyond a mere increase in the number of threats: we are observing the emergence of a complex digital underground economy deeply connected with the malware phenomenon [36, 48]. Nowadays, malware is an essential link of a network of fraudulent activities whose goal is to steal any valuable information from the victims (e.g., credit card numbers, bank accounts, e-mail addresses) and to trade this information at the underground market.

This lucrative motivation had important consequences for the evolution of malicious software. Basically, today's malware is characterized by a steep increase in the sophistication of its code, as authors continuously introduce additional features and specific modifications to maximize the resilience of their creations against anti-malware solutions. Encryption [123], polymorphism [81], metamorphism [128], and other code obfuscation techniques [66, 134] are widely used by malware authors to evade detection. Strictly speaking, malware samples are now able to mutate their syntactical representation after each infection, thus making signature-based strategies completely ineffective. Moreover, to further conceal their presence inside the systems, malware started to include kernel-level components (*root-kits*) to alter the functionalities of the operating system running on the machines of the victims [51, 6]. Such kernel modules are typically used to hide various kind of information (e.g., running processes, open files, and network connections) that could be considered as a symptom of the infection.

Traditional signature-based products are not able to face such an increase in the complexity of malicious code [26, 20]. For this reason, security vendors and researchers are investigating novel analysis and detection strategies. The current trend is to move towards *behavior-based* solutions: according to this paradigm, a suspicious program should be considered malicious because it *exhibits a malicious*

behavior, independently from its actual binary representation. This approach is rapidly becoming the primary method for security labs to automatically understand the behaviors that characterize each new piece of malware and to develop the appropriate countermeasures [5, 141, 120]. On the other side, this technology is also used on end-users' hosts, to monitor the execution of suspicious programs and try to detect and block malicious behaviors in real-time [85, 89, 107].

Behavior-based approaches can be typically classified into two categories, depending on whether they adopt a *static* or a *dynamic* strategy. Static solutions reason on the binary code of a suspicious application without actually running it, so they can consider all the behaviors the program may exhibit at run-time [19, 60, 98]. However, the application of static analysis techniques to malicious software poses serious theoretical problems that limit the overall precision of the final results [63]. For this reason, most of today's approaches are dynamic [5, 76, 141]. Dynamic solutions focus on an actual execution of the target program, they can provide more accurate results, and are more resilient to code obfuscation. To observe the behavior of the suspicious program, dynamic analyzers execute the untrusted application in a special environment, typically a virtual machine, that provides fine-grained monitoring capabilities.

Dynamic behavior-based solutions are very promising, but they are not free from limitations. In particular, state-of-the-art approaches suffer four fundamental problems: they are *incomplete*, *non-transparent*, cannot be used on a system that *has already been compromised*, and they introduce a *non-negligible run-time overhead*. In the following paragraphs we briefly discuss these limitations.

First, all dynamic approaches are incomplete, as they can reason only on a limited number of program paths (i.e., the ones observed during the executions of the program). In other words, if a malware sample does not exhibit its malicious behavior during the observed executions, then it will be considered as benign. Unfortunately, malware is becoming increasingly specialized, as it often targets very specific classes of users and systems. It is not uncommon to see malware that behave maliciously only when specific conditions are satisfied. For example some samples start to log pressed keys only when a user visits the web site of its bank, while others steal the serial numbers of various software programs, but behave innocuously if these applications are not found on the infected machine. Current solutions address the incompleteness of the analysis by systematically exploring all environment-dependent program paths [80, 7], but this is not always possible, and several techniques are already known to thwart these systems [13, 115].

Another limitation of dynamic approaches is that malware is often able to detect the virtual environments they use to monitor the execution of suspicious programs. Indeed, in order to be employed for malware analysis, a virtual environment should not only provide bulletproof separation between the host and guest systems, but it should also operate transparently. In other words, the program running inside the guest should never be able to detect that it is not running on a native machine. In our previous work we proved this is not always the case [74, 75], and further research demonstrated that a malicious program

can leverage many discrepancies between emulated and native environments to detect if it is being analyzed [87]. Despite the fact that recent research work has introduced some platforms that cannot be easily detected by malicious programs [29, 99], these approaches assume the target system is already running inside a virtual machine, so they cannot be used directly on end-users' hosts to implement detection or remediation solutions.

Moreover, it should be considered that writing a perfect malware detector is not just very challenging, but is rather an undecidable problem [25]. In other words, it is not always possible to act proactively, i.e., to detect threats before they infect the system. In some situations, post-infection countermeasures remain the only solution to get rid of a malware and of the damages it may have caused to the system, other than reinstalling the entire machine. Unfortunately, the analysis of an infected system is not so straightforward: once a malicious program has taken the control of the host, it may tamper with the results of the analysis in order to hide its presence. Current approaches typically do not consider this situation, and they suppose the target host has not been compromised, or that the malware cannot tamper with the execution of the analysis tool.

Finally, today's dynamic approaches suffer a high computational overhead that still allows to deploy them in analysis environments, but prevents them from being employed at end hosts.

Miscreants are aware of the recent advances in malware countermeasures, as well as of the aforementioned problems that limit their effectiveness. Indeed, malware authors continuously update their creations to exploit the weaknesses of current anti-malware approaches. We will use the term *next-generation malware* to indicate this emerging class of malicious programs that use sophisticated techniques to circumvent state-of-the-art security products. Obviously, the problems that affect dynamic analyzers are exacerbated when they have to deal with next-generation malware.

1.1 Dissertation contributions

This dissertation provides novel solutions that address the shortcomings of dynamic behavior-based analyzers when dealing with next-generation malware. In particular, our goal is to focus on the problems that still limit the applicability of behavior-based analysis techniques directly at the end hosts, with the intent to provide better protection for the users.

For this purpose, we first present *a framework for improving the completeness of the behavior-based analysis of suspicious programs*. As we already pointed out, next-generation malware often manifest their malicious behaviors only when very specific conditions are met. However, such conditions are more likely to be satisfied on the machines of the end-users rather than in security laboratories, where malicious code is typically analyzed in very synthetic environments. As a consequence, the results of the analysis are very likely to be incomplete. Our

infrastructure allows an end-user to delegate security labs the execution and the analysis of a program, but to force the program to behave as if it were executed directly in the environment of the former. With this approach, the end-user benefits from fine-grained analysis that would be computationally infeasible on his system. On the other hand, the security lab can observe the behavior the program manifests in the execution environment of the end-user. In other words, the security lab can exploit the heterogeneity of users' environments to improve the completeness of the analysis.

Our second contribution is *a generic framework that provides a programming interface to perform complex dynamic system-level analyses of deployed production systems*. By leveraging hardware support for virtualization available nowadays on all commodity machines, the infrastructure we propose (I) is completely transparent and isolated from the system under analysis, (II) does not require any modification to the internals of the system being monitored, and (III) can be installed and removed as the target system runs. As we will show, these characteristics make the framework very suitable for being employed as the basic building block for transparent anti-malware solutions. Moreover, as this approach can deal with both user- and kernel-level code, it can also be used to analyze malware that include root-kit components.

The contributions discussed so far present an important limitation that prevents them from being directly used at end hosts: if the system has already been compromised, the malicious software could tamper with the execution of the analysis or detection tool and hide its presence on the machine. For this reason, we introduce *a software-based attestation scheme for tamper-proof code execution on untrusted hosts*. Our solution guarantees that an arbitrary piece of code can be executed untampered in an untrusted system, even in the presence of malicious software.

Finally, we leverage our virtualization-based analysis framework and our attestation primitive to design a secure solution for the *live and trustworthy forensic analysis of potentially compromised machines*. Our framework can be installed and removed without the need to reboot the machine, and it is completely transparent to an attacker that controls the system. Moreover, the analysis can be periodically and safely interrupted to resume the normal execution of the system.

To summarize, we make the following contributions:

(ICISS 2009) a framework for improving behavior-based analysis of suspicious programs. Our framework allows an end-user to delegate security labs the execution and the analysis of a program and to force the program to behave as if it were executed directly in the environment of the former [72].

(ASE 2010) An infrastructure that provides a programming interface to perform complex dynamic system-level analyses of deployed production systems. We leverage hardware support for virtualization, available nowadays on all commodity machines, to realize a framework that is completely transparent to the system under analysis and also guarantees isolation of the

analysis tools running on top of it [33].

(DIMVA 2010) A software-based attestation scheme for tamper-proof code execution on untrusted legacy systems. The solution we propose provides load-time attestation of a piece of code and also ensures run-time code integrity. Our primitive is resistant to static and dynamic attacks that are known to defeat state-of-the-art approaches [73].

(RAID 2010) A framework to securely perform live forensic analyses of potentially compromised production systems. We demonstrate that this infrastructure is particularly valuable for analyzing an alleged infected system a posteriori, i.e., after the infection took place [71].

1.2 Dissertation organization

The dissertation is organized as follows. In Chapter 2 we describe our framework to allow security labs to observe the execution of suspicious programs in multiple realistic end-users' environments, in order to improve the completeness of the analysis. Chapter 3 presents our generic infrastructure to perform complex dynamic analyses with the guarantee that even kernel-level malware cannot detect the analysis infrastructure. Chapter 4 introduces Conqueror, a software-based code attestation primitive resilient to both static and dynamic attacks. Then, in Chapter 5 we propose a framework to perform live forensic analyses of potentially compromised production systems. Chapter 6 compares our research work with the related literature. Finally, Chapter 7 proposes some possible directions for future work, while Chapter 8 concludes the dissertation.

2

Malware analysis in the cloud

Two of the major disadvantages of dynamic behavior-based analysis are incompleteness and non-negligible run-time overhead. Security laboratories analyze new malicious programs automatically in special environments (e.g., virtual machines) which allow very fine grained monitoring of the behavior of the programs. The automatic behavioral analysis of specialized malware becomes more and more difficult because the malicious behaviors manifest only in very specific circumstances [3]. As an example, the Bancos trojan behaves like a malware only when it runs in the system of a user of a Brazilian bank, but it is innocuous on the vast majority of other systems [32]. If the behavioral analysis of such malware samples is performed in inappropriate environments, like the synthetic ones used in security labs, the results are very likely to be incomplete. On the other hand, if the malicious program were analyzed directly on an end-user's machine, which is the intended target of the attack, the malicious behavior would have more chances to be triggered and it would be caught as it manifests. Unfortunately, the strict lightweight constraint required for end-users' systems does not allow a fine grained analysis of the behaviors of the programs [76, 141]. Consequently, some malicious behaviors (e.g., the leakage of sensitive information) cannot be detected on end-users' machines. Current solutions address the incompleteness of dynamic analysis by systematically exploring all environment-dependent programs paths [80, 8, 137]; however, this is not always possible [13, 115].

In this chapter we propose a new framework for supporting dynamic behavior-based malware analysis, based on cloud computing, that blends together the computational power available in security labs (the cloud) with the heterogeneity of end-users' environments. The rationale of the framework are the two following assumptions. First, the security lab has no limit on the computational resources available and can exploit hardware features, in combination with recent advances in research, to further improve its computational capabilities [15, 83, 50]. Second, end-users' environments are more realistic and heterogeneous than the synthetic environments typically available in security labs and consequently are

better suited for analyzing potentially malicious software. The proposed framework allows an end-user to delegate a security lab the execution and the analysis of an alleged malicious program, and to force the program to behave as if it were executed directly in the environment of the former. The advantage is twofold. It allows the security lab to monitor the execution of a potentially malicious program in a *realistic end-user's environment* and it allows end-users to raise their level of protection by leveraging the computational resources of the security lab for fine-grained analysis that would not be feasible otherwise. Since each end-user's environment differs from the others and since the behavior of a program largely depends on the execution environment, through our framework the security lab can improve the completeness of the analysis by observing how a program behaves in *multiple realistic end-users' environments*. Such in the cloud execution is made possible by a mechanism we have developed for forwarding and executing (a subset of) the system calls invoked by the analyzed program to a remote end-user's environment and for receiving back the result of the computation. As the execution path of a program entirely depends on the output of the invoked system calls, the analyzed program running in the security lab behaves as if it were executed directly in the environment of the user. It is worth pointing out that the solution we propose in this chapter is not a malware detector, but is rather a framework that enhances the capabilities of existing dynamic behavior-based detectors. Examples of malware detectors that could integrate our approach are TTAnalyze [5], Panorama [141], CWSandbox [138], and the layered architecture described in [76].

To evaluate the proposed approach, we have implemented a prototype for Microsoft Windows XP. We used this prototype to study the benefits in term of completeness given by the analysis in multiple execution environments and to measure the performance impact. Our evaluation witnessed that the distributed execution of programs is possible and the computational impact on end-users is negligible. With respect to the traditional analysis in the security lab, the analysis of malicious programs in multiple execution environments resulted in a significant relative improvement of the code coverage: with just four additional distinct end-users' environments we achieved an improvement of $\sim$15%.

To summarize, in this chapter we make the following contributions.

1. We propose a new framework for dynamic behavior-based malware analysis in the cloud.

2. We describe the design and implementation of a working prototype of the aforementioned framework, that has also been integrated into an existing behavior-based malware detector.

3. We perform an evaluation of the proposed framework, demonstrating the feasibility and the efficacy of our idea.

2.1 Overview

Imagine a malicious program, like the one shown in Figure 2.1, that resembles the behavior of the BANCOS malware [32]. To ease the presentation we use high-level APIs of Microsoft Windows; nevertheless our approach works directly with the system calls invoked by these functions. The program polls the foreground window to check whether the user is visiting the website of a Brazilian bank. The existence of such a window is the *trigger condition* of the malicious behavior. If the bank website is visited, the program displays a fake authentication form to tempt the user to type his login and password. Finally, the program forwards the stolen credentials to a remote site.

The automatic analysis of such a piece of malware in a *synthetic execution environment*, like those available in a security lab, is very likely to give incomplete results. Such an environment is generated artificially and consequently it cannot satisfy all the possible trigger conditions of malicious programs. Furthermore, some malicious programs expect inputs from the user and then behave accordingly. As the analysis is performed automatically, user inputs are also artificial and that can prevent the triggering of certain behaviors. On the other hand, we have *realistic execution environments*, the systems of the end-users, which are more suited for analyzing a piece of malware like BANCOS, as they are the intended victims of the malicious activity. Indeed, in the system of a certain class of users, the users of Brazilian banks, our sample malicious program would manifest all its behaviors. Unfortunately, although such systems are more suited for the analysis, it is not reasonable to expect to use all their resources for detecting and stopping potentially malicious programs (fine grained analysis can introduce a slowdown by a factor of 20 [96, 141, 84]). Consequently, host-based detectors perform only very lightweight analysis and cannot detect certain malicious behaviors (e.g., to detect that sensitive information is being leaked using data-flow analysis).

2.1.1 Delegating the analysis to the cloud

In our framework the behavior-based analysis of a new suspicious program is performed in the cloud: the user U does not run directly on his system the suspicious program, nor the malware detector, but he requests the security lab L to analyze the program on his behalf; in turn the latter requests the help of the former to mitigate the fact that its execution environment is synthetic. Our approach to overcome the limitations of the execution environment of L is based on the following assumption: a program interacts with the environment by invoking system calls, and the execution path taken by the program entirely depends on the output of these calls [46]. In our particular context, this assumption means that the triggering of a malicious behavior entirely depends on the output of the system calls invoked. It follows that, to achieve our goal, it is sufficient to force the system calls executed by the program in L to behave as if they were executed

```
VirtualAlloc();
...
VirtualFree();
while (true) {
  hwnd1 = GetForegroundWindow();
  title = GetWindowText(hwnd1);

  if (title == "Banco do Brasil" ||
      title == "Banco Itau" || ...) {
    // Display a fake login screen for
    // the site
    hwnd2 = CreateWindow(...);
    ...
    // Send credentials to a remote site
    socket = WSAConnect();
    WSASend(socket, ...);
    ...
    break;
  }

  Sleep(500);
}
```

Figure 2.1: Pseudo-code of a sample malicious program that resembles the BANCOS trojan.

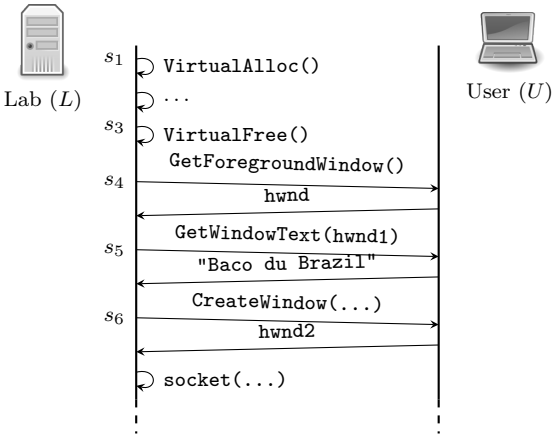

Figure 2.2: Diagram of the execution of the sample malicious program in the security lab (L), by forcing the program to behave as in the environment of the end-user (U).

in U. To do that, the system calls, instead of being executed in L, are executed in U, and L simulates their execution by using the output produced by U. It is worth noting that only a small subset of all the system calls executed by the program might actually affect the triggering of a malicious behavior. Examples of such system calls are (I) those used to access user's data (e.g., the file system and the registry), (II) those used to query particular system information (e.g., active processes, system configuration, open windows), and (III) those used to interact with the users (e.g., to process keyboard and mouse events). Therefore, the collaboration of U is needed only for these system calls, while the remaining ones can be executed directly in L, where the computational resources available allow more sophisticated analyses.

Figure 2.2 shows how our sample malicious program is executed and analyzed leveraging our framework. The scenario of the analysis is the following. The user U has received a copy of the program by e-mail (or by another vector) and he executes the program. With a conventional behavior-based detector the program would be analyzed entirely on the host. With our framework instead, the program is not executed locally but it is submitted to the security lab L, that executes and analyzes the program with the cooperation of the user. The new analysis environment thus becomes $\langle L, U \rangle$. All the system calls executed by the program are intercepted. Our sample program initially executes some system calls $s_1, \ldots, s_3$ whose output does not depend on the environment (e.g., to allocate memory). These system calls are executed directly in L. Subsequently, the program tries to detect whether the user is browsing a certain website: it invokes s_4 = `GetForegroundWindow` to get a reference to the window currently

active on the desktop of the user. As the output of this call highly depends on the execution environment, L requests U to execute the call: L forwards s_4 to U, U executes s_4 and sends back the output to L. The program does not notice what is happening in the background and continues the execution. The next system call is $s_5 =$ `GetWindowText`, which is used to get the title of the foreground window. As one of its input arguments (`hwnd1`) is the output of a system call previously executed in U, s_5 is also executed in U. Supposing that the user in U is actually visiting a website targeted by the program, the trigger condition is satisfied and the program displays the fake login form to steal the user's credentials. As this activity involves an interaction with the user and such interaction is essential to observe the complete behavior of the program, the system calls involved with this activity are also forwarded to U, to get a realistic input. L can eventually detect that there is an illegitimate information leakage.

The in the cloud execution of a potentially malicious program does not expose the end-user to extra security risks. First, we confine the dangerous modifications the program could make to the system in the environment of the security lab. Second, more malicious behaviors can be detected and stopped, because the analysis performed in the lab is more thorough. Third, the execution of the program consumes less resources, as the user is in charge of executing a subset of all the system calls of the program. Fourth, annoying popups are still redirected and shown to the user, but that would happen also if the program were executed normally.

2.1.2 Exploiting diversity of end-users' environments

The proposed framework allows to monitor the execution of a potentially malicious program in multiple execution environments. Given the fact that end-users' environments are very heterogeneous (e.g., users use different software with different configurations, visit different websites), it is reasonable to expect that the completeness of the analysis improves with the increase of the number of different environments used.

To analyze a program in multiple execution environments, it is sufficient to run multiple instances of the analyzer, $L_1, \ldots, L_n$, such that each instance cooperates with a different environment $U_1, \ldots, U_n$ to execute the system calls that might affect the triggering of the malicious behaviors (i.e., the environments used are those of n of the potential victims of the malicious program, chosen according to some criteria). The security lab can thus observe how each analysis environment $\langle L_i, U_i \rangle$ affects the behavior of the program and can merge and correlate the behaviors observed in each execution. It is worth pointing out that each environment U_i is involved in the analysis only when it tries to execute a suspicious application; on the contrary, an end-user environment that does not execute a potentially malicious program is never intentionally exposed to unwanted threats.

Figure 2.3 shows how the analysis of our sample program is performed simultaneously in multiple execution environments $\langle L_1, U_1 \rangle, \ldots, \langle L_6, U_6 \rangle$. Each execu-

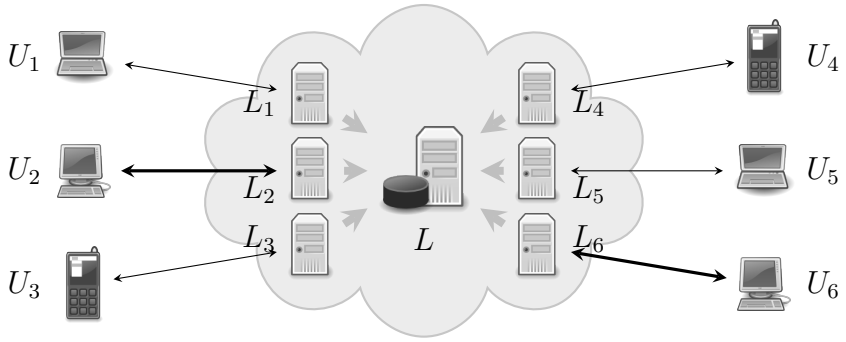

Figure 2.3: Diagram of the execution of multiple instances of the analysis of a suspicious program in multiple execution environments $\langle L_1, U_1\rangle, \ldots, \langle L_6, U_6\rangle$. The central entity L aggregates the results of each analysis.

tion is completely independent from the others but the results of the analysis are collected and correlated centrally by L. As $U_1, \ldots, U_6$ are distinct environments, we expect the forwarded system calls to produce different output (e.g., to return different window titles) and thus to cause the various instances of the analyzed program to follow different paths. In the example, we have that the trigger condition is satisfied only in U_2 and U_6, but the websites being visited are different (one user is visiting the website of "Bancos do Brazil" and the other one the website of "Banco Itau"). Therefore, the correlation of the results reveals that the program is effectively malicious and some of its trigger conditions.

2.2 Design and implementation

The two parties participating in the in the cloud analysis of a program are the security lab, L for short, and the end-user (the potential victim), U for short. In this section we describe the components we have developed for these two parties to make such distributed execution possible. The current prototype implementation is specific for Microsoft Windows XP, but the support for other versions of the OS can be added with minimal efforts. At the moment, our prototype can successfully handle all the system calls involving the following system resources: file, registry keys, system and processes information, and some graphical resources.

2.2.1 Executing a program in multiple environments

The execution of a suspicious program in multiple environments presents several problems. In the following we discuss the challenges we had to face during the development of our analysis infrastructure, together with the solutions we propose.

System calls hooking

To intercept the system calls executed by the analyzed program, we leverage a standard user-space hooking technique. We start the process we want to monitor in a suspended state and then inject a DLL into its virtual address space. The DLL hooks the functions `KiIntSystemCall` and `KiFastSystemCall`, two small function stubs used by Microsoft Windows for executing system calls [51, 102]. This approach allowed to simplify the development and facilitated the integration of the framework into an existing malware detector.

System calls proxying

A user-space application cannot directly access the data structure representing a particular resource of the system (e.g., a file, a registry key, a mutex, a window) but it has to invoke the appropriate system calls to obtain an opaque reference, a *handle*, to the resource and to manipulate it. We exploit this characteristic of the operating system to guarantee a correct functioning of the analyzed program, and to simulate the existence of resources with certain properties that exists on a remote system, but do not in the system in which the program is executed. When a system call is invoked, we analyze the type of the call and its arguments to decide how to execute it: locally or remotely.

To differentiate between *local* and *remote* calls, we check if the system call creates a handle or if it uses a handle. To create a handle means to open an existing resource or creating a new one (e.g., to open a file), while to use a handle means to manipulate the resource (e.g., to read data from an open file). In the first case, we analyze the resource that is being opened and according to some rules (details follow) we decide whether the manipulation of the resource might influence the triggering of a malicious behavior. If not, we consider the resource and the system call *local* and we execute the call in L. Otherwise, we consider the resource and the system call *remote* and we forward and execute the latter in U. When we intercept a system call that uses a handle, we check whether the resource being manipulated (identified by the handle) is local or remote and we execute the call in L or U accordingly.

Figure 2.4 represents the various components we have developed (highlighted) to intercept system calls and to execute them either locally or remotely. All system calls executed by the analyzed program P are intercepted. Local system calls are passed to the kernel as is, remote ones are forwarded to the system of the end-user. To execute a remote syscall in U, L serializes the arguments of the system call and sends them to U. The receiver deserializes the arguments, prepares the program state for the execution (i.e., by setting up the stack and the registers), and then executes the call. When the syscall returns, U serializes the output arguments and sends them back to L. Finally, L deserializes the output arguments, where the program expects them, and resumes the normal execution. The program P cannot notice when a system call is executed elsewhere, because it finds in memory the expected output.

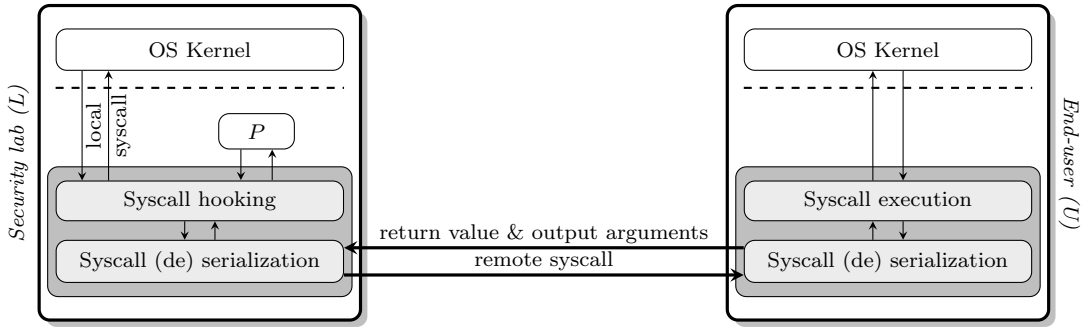

Figure 2.4: System calls interception and remote execution (P is the analyzed program)

There are certain types of handles that do not represent a tangible resource (e.g., a file, a registry key, or a mutex), but instead represent resources that are used to execute other system calls asynchronously. When such a handle is created we do not know if it will be used to synchronize (or notify) a local or a remote system call. Our solution is to consider the system call and the resource it creates as *lazy*. Thus, if a system call is lazy, we simulate its execution and we postpone the real execution until the resource it creates is used by another system call, and we can tell whether the handle will be used locally or remotely. To safely postpone the execution it is necessary to save the arguments, because they might be overwritten by subsequent operations, and restore them at the proper time. Arguments are saved in memory using the same serialization primitives we use for remote execution.

On paper, the mechanism for serializing and proxying a system call looks simple; however, its implementation is very challenging. The Microsoft Windows system call interface, known as *native API*, is poorly documented and may change due to operating system updates. We put a lot of reverse engineering efforts to understand how to properly serialize all system calls and their arguments. After all, the Windows native API turned out to be well suited for proxying and to simulate the existence of resources that physically reside on a different system. No system call can operate concurrently on two resources, resources can always be distinguished, and system calls manipulating the same resource are always executed in the same environment.

In-situ (de)serialization

A prerequisite for transparently proxying a system call is that, at the return of the call, the program must find in memory the output exactly as if the output were produced locally. Thus, the consistency of the data-structures representing the various arguments of the system call must be preserved. Unfortunately, the (de)serialization of system call arguments is very challenging: many arguments contain pointers and others are completely opaque (i.e., huge arrays of bytes that conceal very complex data structures and sometimes contain internal pointers).

We have addressed the problem of serializing such complex data-structures by

developing a generic mechanism we called *in-situ (de)serialization*. In short, the mechanism consists in maintaining constant the locations at which the various arguments are stored and in maintaining constant the value of the pointers they contain. The advantage is that we can guarantee consistent pointers with no effort because we never touch their values. A requirement for in-situ (de)serialization is that, in U, the memory locations at which the arguments have to be deserialized must be accessible and must not contain program data. To satisfy this requirement, we "align" the address space of the module running in U, with the address space of the monitored program in L. The alignment consists in reserving in the user component all the memory pages that the monitored program could use to store system calls arguments and by committing these pages on-demand. That prevents the user component to use the aforementioned memory pages for anything but the in-situ (de)serialization.

Choosing remote system calls

Remote system calls are selected using a whitelist. The whitelist contains a list of system call names and a set of conditions on the arguments. Examples of the system calls we consider remote are: `NtOpenKey`, `NtCreateKey` (if the arguments indicate that the key is being opened for reading), `NtOpenFile`, `NtCreateFile` (if the arguments indicate that the file is being opened for reading), `NtQuerySystemInformation`, and `NtQueryPerformanceCounter`. The handles returned by these calls are flagged as remote, by setting the most significant bits (which are unused)[1]. Thus, we can identify subsequent system calls that access a remote resource and we have the guarantee that no overlap between handles referencing local and remote resources can occur. Even when we execute a system call on U because one of its arguments is flagged as remote, we still match the system call against our whitelist to guarantee that the environment of the end-user is not accidentally exposed to security threats.

GUI system calls

User's inputs and GUI resources often represent trigger conditions. For this reason it is important to let the analyzed program to interact with realistic user's inputs (i.e., GUI events) and resources. Although in Microsoft Windows all the primitives of the graphical user interfaces are normal system calls, to facilitate the proxying, we rely on the Windows Terminal Services subsystem to automatically forward the user interface of the monitored application from the lab to the user's machine. In particular, our prototype uses *seamless* RDP (Remote Desktop Protocol) [14], that allows to export to a remote host the graphical interface of a single application instead of the entire desktop session. Therefore, if the analyzed program executed in the lab displays the user a fake login form and

[1] As concrete handle values are opaque to user-space applications, by manipulating their unused bits we do not alter the semantics of the process being monitored.

blocks for inputs, the form is transparently displayed in U and the received user's events (keystrokes and mouse clicks) are sent back to the program running in L.

The solution based on RDP allows only to forward a GUI to a remote system. However, the session in which the application is run belongs to L. Thus, attempts to query the execution environment would return the status of the environment in L. As an example let us consider the system calls associated with the API functions `GetForegroundWindow` and `GetWindowText`, used by our sample malware (Figure 2.1) to check if the victim is visiting the website of a Brazilian bank. Without any special handling these system calls would return the windows of the session (on L). We want instead these calls to return information about the windows found in the remote environment. To do that, we execute them remotely as any other remote system call.

Memory-mapped files

Memory-mapped files violate the assumption that system resources can be manipulated only through system calls. Our approach to handle memory mapped files is as follows. We intercept and forward the system calls used to map a file into memory to U. In L instead we reserve, and protect from any access, the memory regions at which the file is mapped. When the analyzed program tries to read, write, or execute data from the memory mapped file, a page fault exception occurs. We intercept the exception, retrieve the page accessed by the program from U, and we store the content of the page locally at the same address. Finally, we update the permissions of the page to authorize future accesses and resume the execution of the program. The program will transparently continue its execution. To intercept page faults we have developed a kernel driver that we install in L to replace the default exception handler. The kernel driver is necessary to bypass the Windows Structured Exception Handling (SEH) mechanism and to prevent the execution of several system calls that were executed (some of which could be considered remote) before the exception would be caught by our system.

One-way isolation

One of the goals of our framework is to protect the system of the end-user from damages that could be caused by the analyzed program, without interfering with the execution of the program. The approach we adopt to achieve this goal is based on one-way isolation [124]: "read" accesses to remote system resources are allowed, but "write" accesses are not and are performed locally. That is, if the program executes a system call to create or to modify a resource we normally consider remote, we treat the resource as local and do not proxy the call. To guarantee a consistent program state, we also execute locally all subsequent system calls involving such resource.

In case the analyzed program turned out to be benign, system changes made in the lab environment could be committed to end-user's environment. Our prototype currently does not support this feature, nor does it support the correct

isolation of a program that accesses a resource that is concurrently accessed by another.

2.2.2 An in the cloud behavior-based malware detector

In order to demonstrate how our framework can naturally complement behavior-based malware detectors, we have integrated it in an existing detector [76], which is based on virtual machine introspection and is capable of performing fine grained information flow tracking and to identify data-flow dependencies between system call arguments. The malware detector is built on top of a customized system emulator, which supports system calls interception and taint analysis with multiple taint labels. As our framework works directly inside the guest, the integration of the two components required only a trivial modification to allow the detector to isolate the system calls executed by the suspicious program from those executed by our prototype to proxy system calls and to ignore the latter.

To monitor the execution of a suspicious program in multiple end-users' environments it is sufficient to run multiple instances of the enhanced malware detector just described, where each instance collaborates with a different end-user's machine, and to merge the results. We have not yet addressed the problem of correlating the results of multiple analyses.

2.3 Evaluation

This section presents the results of testing our prototype implementation of the framework and presents a conceptual comparison of our approach with existing solutions that try to systematically explore all program paths. We evaluated the prototype with benign and malicious programs. The results of the evaluation on benign programs witness that our approach does not interfere with normal program execution and that it introduces a negligible overhead. Moreover, the evaluation demonstrates that the analysis of a piece of malware in multiple execution environments significantly improves the completeness of the results: with the collaboration of *just four* different execution environments we observed a $\sim$15% relative improvement of the code coverage.

2.3.1 Experimental setup

The infrastructure used for the evaluation corresponds to the one described in Section 2.2.2, with the difference that, instead of performing behavior-based detection, we tracked the basic blocks executed in each run of the experiments. To simulate the lab environment we used a vanilla installation of Windows XP running inside the emulator, while as users' environments we used some other machines and we acted as the end-users.

Program	Action	Local	Remote
ClamAV	Scan (remote) files with (remote) signatures	166,539	1,238
Eudora	Access and query (remote) address book	1,418,162	11,411
Gzip	Compress (remote) files	19,715	93
MS IE	Open a (remote) HTML document	1,263,385	10,260
MS Paint	Browse, open, and edit (remote) pictures	1,177,818	9,708
Netcat	Transfer (remote) files to another host	16,007	93
Notepad	Browse, open, and edit (remote) text files	929,191	7,598
RegEdit	Browse, view, and edit (remote) registry keys	1,573,995	13,697
Task Mgr.	List (remote) running processes	33,339	241
WinRAR	Decompress (remote) files	71,195	572

Table 2.1: List of tested benign programs, actions over which each program was exercised, and number of locally and remotely executed system calls (GUI system calls are not counted).

2.3.2 Evaluation on benign programs

To verify that our framework did not interfere with the correct execution of the programs, we executed through our prototype multiple benign applications. The tested programs included both command-line utilities and complex GUI applications. Table 2.1 reports the set of programs tested, together with the actions over which each program was exercised and with the number of local and remote system calls. We interacted with each program to perform the operations reported in the table. As we ran the experiments with the proxying of all supported system calls enabled, the numbers in the table indicate the total number of remotely executed calls and not only those involved with the described actions. For example, we used ClamAV to scan all the content of a directory. Through our framework the anti-virus transparently scanned a directory existing only in the simulated end-user's system, using a database of signatures which also existed only in the remote system.

We successfully executed all the actions reported in the table and verified that the resources that were accessed effectively corresponded to those residing on the system of the end-users. The number of system calls executed indicates that the programs used for the evaluation are quite complex and thus that our results are good representatives. We can conclude that: (I) system calls accessing remote resources do not interfere with system calls accessing local resources, (II) our framework does not interfere with the correct execution of programs, and (III) system calls proxying allows to transparently access system resources residing on remote hosts.

2.3.3 Performance overhead

We used a subset of the benign programs of Table 2.1 to evaluate the overhead introduced by our framework on the systems of the user and of the security lab. We observed that the number of remotely executed system calls depended on the type of applications and the actions exercised; consequently the overhead depended on these factors. On the system of the end-user, we measured a CPU, memory, and network usage that was roughly proportional to the number of remotely executed system calls. Nevertheless, in all cases, the resources consumed never exceeded the resources consumed when the same programs were executed natively on the system: on average we observed a 60% and 80% reduction of CPU and memory usage respectively. On the other hand, we noticed a slight increase of the resource usage in the system in the lab: on average we observed a 36% and 77% increase of CPU and memory usage respectively. We also measured that, on average, 956 bytes have to be transferred over the network to remotely execute a system call. For example, the execution of RegEdit required in total to transfer 1030Kb of data. In conclusion, our framework has negligible performance impact on the end-user and the impact on the security lab, without considering the overhead introduced by the analysis run on the framework, is sustainable and can be drastically reduced by improving the implementation (e.g., by compressing data before transmission).

2.3.4 Evaluation on malicious programs

We evaluated our framework against multiple malicious programs representing some of the most common and recent malware families. The goal of the evaluation was to measure whether the analysis of multiple executions of the same piece of malware, in different end-users' environments, gives more complete results than the analysis of a single execution of the program in an unrealistic environment (i.e., the vanilla installation of Windows XP).

To quantify the completeness of the results we measured the increase of code coverage. We initially executed batch (i.e., without any user interaction) each malicious program in the environment of the security lab and we recorded the set of unique basic blocks executed (excluding library code). Subsequently, we ran each malicious program multiple times through our prototype, each time in collaboration with a different end-user's environment, and again we recorded the set of unique basic blocks executed. Therefore, if b_0 represents the set of basic blocks executed in the environment of the security lab, and $b_i, i > 0$, represents the set of basic blocks executed with the collaboration of the i^{th} end-user's environment, the increase of code coverage after the i^{th} execution is measured as $|b_i \setminus (b_{i-1} \cup \ldots \cup b_0)|$.

Figure 2.5 reports the relative increase of code coverage (using b_0 as baseline) measured during our evaluation, leveraging just four different end-users' environments and 27 different malware samples. The figure clearly shows that in the

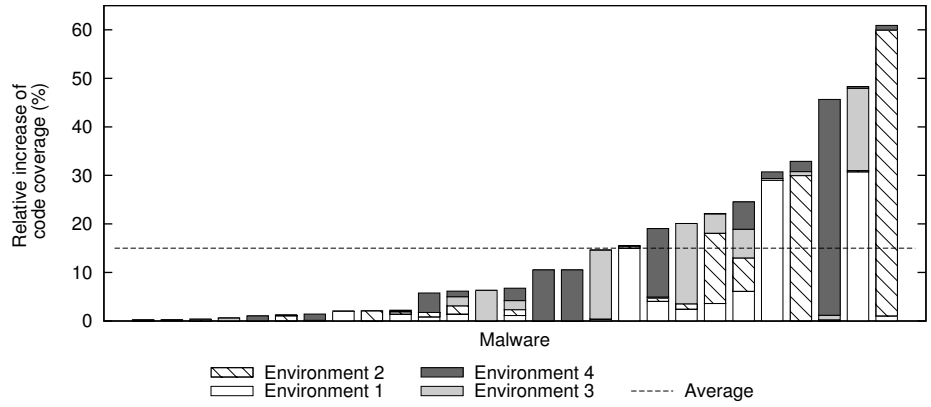

Figure 2.5: Relative increase of code coverage obtained by analyzing the tested malware samples in multiple execution environments.

majority of the cases we have a noticeable relative increase of the code coverage; the average increase is 14.53%, with a minimum of 0.24%, to a maximum of 60.92%. It is worth noting that, although the observed improvements appear minimal, most of the time small percentages correspond to the execution of hundreds of new basic blocks. It is also important to note that certain environments contributed to improve the results with certain malware but did not contribute at all with others. Indeed, the four environments contribute respectively on average 25.35%, 30.86%, 18.14%, and 25.68% of the total increase observed.

For example, during the analysis of a variant of Satiloler, we noticed that the monitoring of web activities was triggered only in one of the four environments, when we visited a particular website. Thus, in this environment we observed a 16.54% increase of the relative code coverage, corresponding to the execution of about 140 new unique basic blocks; the observed increase in the other environments did not exceed 3%. Another example is Antigen, a malware that displays a modal window and blocks its execution until the user clicks a button. With our approach, the modal window is forwarded to the user's machine, and the user promptly clicks on the button, thus dismissing the window and allowing the malware to continue its execution and to manifest its malicious behaviors (i.e., theft of visited URLs, ICQ and dial-up accounts information).

In conclusion, we believe the relative improvements observed during the evaluation testify the effectiveness of the proposed approach at enhancing the completeness of dynamic analysis.

2.3.5 Conceptual comparison with input oblivious analyzers

Input oblivious analyzers are tools capable of analyzing exhaustively a malicious program by systematically forcing the execution of all program paths [80, 8]. When an input-dependent control flow decisions is encountered, both program

branches are explored. Such systematic exploration is achieved by manipulating the inputs and updating the state of the program accordingly, leveraging constraint solvers, to force the execution of one path and then of the other.

The framework we propose in this chapter addresses the same problem through a completely different approach. Although our methodology might appear less systematic, it has the advantage that, by leveraging real execution environments, it can deal with complex trigger conditions that could exhaust the resources of input oblivious analyzers. For example, trigger conditions dominated by a complex program structure might easily generate an unmanageable number of paths to explore and unsolvable constraints. Indeed, several situations are already known to thwart these systems [13, 115]. Examples of other situations that can easily render input oblivious analyzers ineffective are malicious programs with payload delivered on-demand (e.g., the Conficker malware [94]) and programs with hidden malicious functionality, like rogue anti-viruses, where the trigger conditions consist in multiple complex asynchronous events. As we assume that sooner or later the malicious program will start to reap victims, we can just sit and watch what a program does in each victim's system, without being affected by the complexity of trigger conditions. At the first sign of malicious activity, we consider the program as malicious; then we can notify all victims, but we could also continue to analyze the program in some of the affected systems.

2.4 Discussion

The framework we propose can clearly raise privacy issues: by controlling the system calls executed on the system of an end-user, the security laboratory can access sensitive user's data (e.g., files, registry keys, GUI events). We are convinced that the privacy issues introduced by our approach are comparable to already existing issues. For example, commercial behavior-based detectors incorporate functionality, typically enabled by default, to submit to laboratories suspicious executables or memory dumps of suspicious processes (which can contain sensitive user data). Thus privacy of users is already compromised. Moreover, the security laboratory is just a special provider of cloud services: users have to trust it like they trust other providers (e.g., e-mail providers and web storage services).

Moreover, our framework is sensitive to various forms of detection and evasion. As an example, the user-space hooking technique we employ to intercept the system calls issued by a suspicious program can be thwarted very easily. To prevent similar attacks and also others based on the identification of emulated or analysis environments, it would be sufficient to build our framework on top of undetectable systems for malware analysis, such as the one we present in Chapter 3. The limitations of our current implementation (e.g., lack of support for inter-process communication) can also offer opportunities for detection and evasion. We believe the majority of the attacks will not be possible with a complete implementation.

3

Transparent and efficient dynamic analysis

All behavior-based analysis and detection techniques share a common characteristic: to observe a suspicious application, they have to *monitor* its execution. Basically, monitoring can be performed in two ways: by leveraging a user- or system-level module, or using an out-of-the-box approach. Kernel-level malware, which compromise the kernel of an operating system (OS), can easily fool the solutions that adopt the first approach, as the malware is executed at the same privilege level of the OS [51, 6]. For instance, Shadow Walker exploits kernel-level privileges to defeat memory content scanners by providing a de-synchronized view of the memory used by the malware and the one perceived by the detector [122]. To address the problem of kernel-level malware and of attackers that are able to obtain kernel-level privileges, researchers proposed to run out-of-the-box analyses by exploiting virtual machine monitor (VMM), or hypervisor, technology. In such a context, the analysis is executed in a trusted environment, the VMM, while the monitored OS and users' applications are run as a guest of the virtual machine. Unfortunately, today's malicious software often incorporates a variety of tests to detect whether it is executed in a virtual machine, and to obfuscate its behavior if it suspects its execution is being monitored. Even worse, our research demonstrated that such tests can be generated with a fully automatic and systematic methodology [87].

Recently, the introduction of hardware extensions for the x86 architecture made possible to implement more transparent virtual machine monitors [2, 53]. Using such extensions, the hypervisor can operate at a higher privilege level than the guest OS, it has complete control of the hardware, it can preemptively intercept events, it cannot be tampered by a compromised OS, and therefore it can be used to enforce stronger protection [29, 42, 92, 99, 110, 116]. Advanced techniques, like the one used by Shadow Walker to hide malicious code, are defeated using out-of-the-box memory content scanners. All the VMM-based solutions proposed in literature are based on the same assumption: they operate *proactively*. In other words, the hypervisor must be started before the guest OS and it must run until the guest terminates. Therefore, post-infection analysis of systems

that were not running such VMM-based protections before an infection continues to be unsafe, because the malware and the tools used for the analysis run at the same privilege level.

To address these problems, in this chapter we propose a generic framework that allows to perform dynamic and transparent analyses of both user- and kernel-level code in commodity production systems. Our approach does not require to instrument the system under test, thus it can be used also on off-the-shelf products whose source code or debugging symbols are not necessarily available. Moreover, the framework we propose is able to inspect systems running on real hardware, since it does not require an emulation container. Similar to existing out-of-the-box approaches, ours can be used to analyze both the kernel and user-space components. However, differently from existing solutions, ours is *fully dynamic*, *transparent*, *loosely dependent on the operating system*, and *fault-tolerant* with respect to possible defects in the analysis code. First, our framework does not require recompilation or rebooting of the target system. Thus, it can be used to analyze any running production system, including commodity systems lacking native support for instrumentation and systems not running in virtual machines. Second, the framework is not invasive, since analyses can be performed on a virtually unmodified system: as explained in the following, only a minimal driver needs to be installed and no parts of the kernel are patched in any way. Moreover, since the framework itself is not accessible from the target system, its code cannot be detected by malicious programs. Thus, the infrastructure can be applied to any operating system, as the majority of the facilities it supports are completely OS-independent, and the only OS-dependent functionalities are just provided to ease the development of analysis tools built on top of it. Finally, the framework is fault-tolerant, as it guarantees that a defect in an analysis tool does not damage the framework itself, nor the analyzed system.

Our framework leverages hardware extensions for virtualization available on commodity x86 CPUs [2, 82]. Hardware-support for virtualization allows the development of virtual machine monitors that are very efficient, completely transparent, and non invasive to the systems running in the virtual machine. To overcome the major limitation of traditional VMM-based approaches (i.e., the impossibility to analyze productions systems not running in a virtual machine), our framework exploits a feature of the hardware that allows to install a virtual machine monitor and to *migrate a running system into a virtual machine*. When the analysis is completed, the original mode of operation of the system can be restored. Practically speaking, our framework is a minimalistic virtual machine monitor acting as a broker between the analyzed system and the analysis tool. The framework abstracts low-level events occurring in the analyzed system into high-level events and guarantees fault-tolerance by relying on the hardware to run the analysis tool in an isolated execution environment.

To demonstrate the potential of our framework we have developed an interactive kernel debugger, nicknamed HyperDbg, constructed entirely using the programming interface exposed by our infrastructure. HyperDbg adds live and in-

teractive debugging support to Linux and Microsoft Windows XP, so far only possible using very invasive tools, like Syser [127], or traditional VMM-based debuggers. HyperDbg can be used to debug any component of the Linux and Windows kernels, including interrupt/exception handlers, device drivers, and even supports single instruction stepping. Being completely separated from the debuggee, HyperDbg is transparent to the analyzed system, thus it cannot be detected by a malicious program. These characteristics make HyperDbg very suitable to be employed for the interactive analysis of malicious programs, including those that contain kernel-level components. Then, in Chapter 5, we extend the framework to design a more comprehensive solution to perform live forensic analyses of potentially compromised production systems.

We designed our framework with the intent to provide a transparent and efficient infrastructure for the development of new malware analysis and detection tools. However, it is worth pointing out that the solution we propose is generic, and can be used to perform other sophisticated and non-intrusive dynamic analyses on system-level code, including profiling and tracing of the kernel and user-space applications, interactive debugging, or even extension of system features.

In summary, this chapter makes the following contributions.

1. We propose a generic framework to perform complex dynamic system-level analyses of commodity production systems. Compared to existing frameworks, the one we propose guarantees transparency, efficiency, and does not require the target system to be already installed on a virtual machine.

2. We implemented our framework in an experimental prototype for Microsoft Windows XP. Recently we extended our prototype to support also the Linux operating system.

3. We describe the design and the implementation of HyperDbg, a kernel-level interactive debugger built on top of our framework.

Both the analysis framework and HyperDbg are available at `http://code.google.com/p/hyperdbg/` and are released under the terms and conditions of the GPL (v3.0) license.

3.1 Intel VT-x

Before presenting our VMM-based framework, we give a brief overview of the hardware virtualization technology available in Intel x86 CPUs, called VT-x [82]. AMD technology, named SVM [2], is very similar and differs mostly in terms of terminology.

Intel VT-x separates the CPU execution into two modes of operation: *VMX root mode* and *VMX non-root mode*. The VMM and the guest (OS and applications) execute respectively in root and non-root modes. Software executing in

both modes can operate in any of the four privilege levels that are supported by the CPU. Thus, the guest OS can execute at the highest CPU privilege and the VMM can supervise the execution of the guest without any modification of the guest. When a VMM is installed, the CPU switches back and forth between non-root and root mode: the execution of the guest might be interrupted by an *exit* to root mode and subsequently resumed by an *enter* to non-root mode. After the launch, the VMM execution is never scheduled and exits to root-mode are the only mechanism for the VMM to regain the control of the execution. Like hardware exceptions, exits are events that block the execution of the guest, switch from non-root mode to root mode, and transfer the control to the VMM. However, differently from exceptions, the set of events triggering exits to root mode can be configured dynamically by the VMM. Examples of exiting events are exceptions, interrupts, I/O operations, and the execution of privileged instructions that access control registers or descriptor tables. Exits can also be requested explicitly by the guest through a *VMM call*. Exits are handled by a specific VMM routine that eventually executes an enter to resume the execution of the guest.

The state of the CPU at the time of an exit and of an enter is stored in a data structure called Virtual Machine Control Structure, or *VMCS*. More precisely, the VMCS stores the *host state*, *guest state*, and the *execution control fields*. The host state stores the state of the processor that is loaded on exits to root mode, and consists of the state of all the registers of the CPU (except for general purpose registers). Similarly, the guest state stores the state of the processor that is loaded on entries to non-root mode. The guest state is updated automatically at every exit, such that the subsequent entry to non-root mode will resume the execution from the same point. The execution control fields allow a fine-grained specification of which events should trigger an exit to root mode.

In the typical deployment, the launch of the VMM consists of three steps. First, the VMX root-mode is enabled. Second, the CPU is configured to execute the VMM in root-mode. Third, the guests are booted in non-root mode. However, Intel VT-x supports a particular feature, called *late launching of VMX modes*, that allows to launch a VMM at any time, thus giving the ability to transform a running host into a guest of a VMM. The procedure for such a delayed launch is the same as the one just described, with the exception of the third step. The state of the CPU for non-root mode is set to the exact same state of the CPU preceding the launch, such that, when the launch is completed, the execution of the OS and its applications resumes in non-root mode. The inverse procedure can be used to unload the VMM, disable VMX root-mode, and give back full control of the system to the OS.

3.2 Overview of the framework

Figure 3.1 depicts the architecture of our framework, the installation and removal processes, and the migration of the operating system and its applications into a

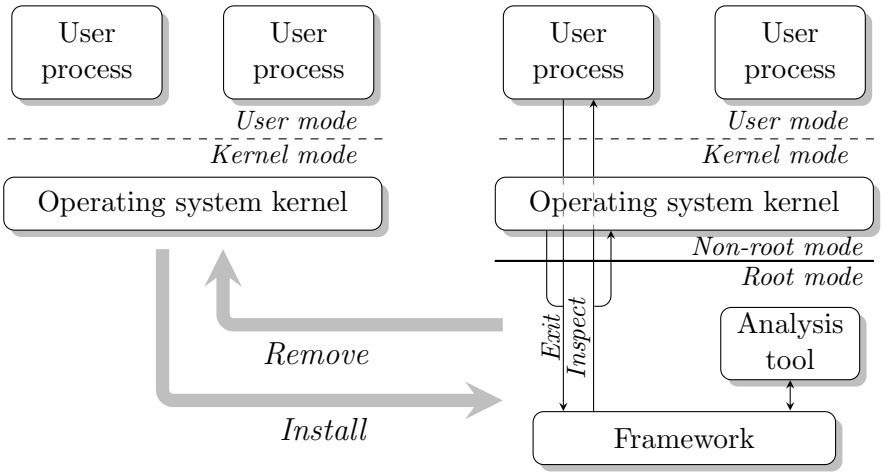

Figure 3.1: Overview of the framework

virtual machine. Our framework consists of a virtual machine monitor that provides a programming interface for the development of system-level analysis tools. As in traditional VMM-based analysis approaches, the analysis tool is run within the VMM and thus completely transparent to guests of the virtual machine. However, compared to traditional VMM-based ones, ours does not require the system to be already running inside any virtual machine. To achieve this goal, our framework leverages hardware extensions for virtualization available on all modern x86 CPUs (which are unused in the majority of the deployments). As discussed in Section 3.1, these extensions augment the instruction set architecture with two new modes of operation: VMX root mode and VMX non-root mode[1]. These new modes of operation separate logically the virtual machine monitor from a guest without having to modify the latter. More precisely, we exploit late launching of VMX modes to install a virtual machine monitor even if the system has already been bootstrapped. In other words, late launching allows to migrate (temporarily) a running operating system in a virtual machine, and to analyze and control the execution of the system from the monitor. Through the rest of the chapter, we use the term "guest" to refer to the system under analysis that has been migrated into a virtual machine.

Practically speaking, the running operating system is not migrated anywhere and not touched at all. Rather, by launching VMX modes, the execution environment is extended with the two aforementioned operating modes; the running operating system is then associated with non-root mode, while the VMM is associated with root mode. Thus, in all respects, the operating system and its applications become a guest of our special virtual machine. Following the same principle, the VMM can be unloaded, and the original mode of execution of the operating system restored, by simply disabling VMX modes. After the launch

[1]VMX (non-) root mode is the terminology used by Intel; AMD adopts a different terminology.

of the VMX modes, the execution of the guest can continue exactly as before, even in terms of interactions with the underlying hardware devices. However, during its execution, the guest might be interrupted by an exit to root mode. Being executed at the highest privilege level, the routine that handles the exit has complete read/write control of the state of the guest system (of both memory and CPU registers).

The framework itself does not perform any analysis. It is only responsible for handling a small set of exits to control all accesses to the memory management unit of the CPU, to prevent the guest from accessing the physical memory locations holding the code and the data of the framework. On the other hand, the framework provides a flexible API to develop tools to perform sophisticated analyses of both kernel and user code running in the guest. Using the functionalities provided through the API, the tool can request the framework to monitor certain events that might occur during the execution of the guest; when such events occur, it can inspect, and even manipulate, the state of the guest. The events that can be monitored include, but are not limited to, system call invocations, function calls, context switches and I/O operations. Practically speaking, events are monitored through exits to root mode. Thus, a request of the analysis tool to monitor a certain high-level event (e.g., the execution of a system call) is translated by the API of the framework into a sequence of low-level operations that guarantee that all the occurrences of such event in the guest trigger an exit to root mode. Similarly, the framework translates the exit into a higher-level event and notifies the occurrence of the event to the analysis tool. Once notified, the tool can recover information about the event (e.g., arguments and return value of a system call), using the inspection functionalities offered by the API.

A fundamental requirement for the analysis of production systems is that analysis tools must not interfere with the correct execution of the guest. This is particularly important for faults and deadlocks that might occur in the analysis tool. The approach we adopt is to run the tool in a less privileged execution environment, isolated from the analyzed system and from the framework. The tool can interact with the guest only through the API exposed by the framework. This approach guarantees the framework the ability to intercept any fault occurring in the tool, to mediate all accesses to the analyzed system (and to prevent write accesses), and to terminate the tool in case of deadlocks or other anomalous situations.

3.3 Design and implementation

Figure 3.2 shows a more detailed view of the architecture of our framework. Intuitively, this architecture is very similar to that of traditional operating systems: the framework plays the role of the kernel and the analysis tool plays the role of a user-space application. As will become clear later, this architecture prevents buggy analysis tools from compromising the guest system and the framework.

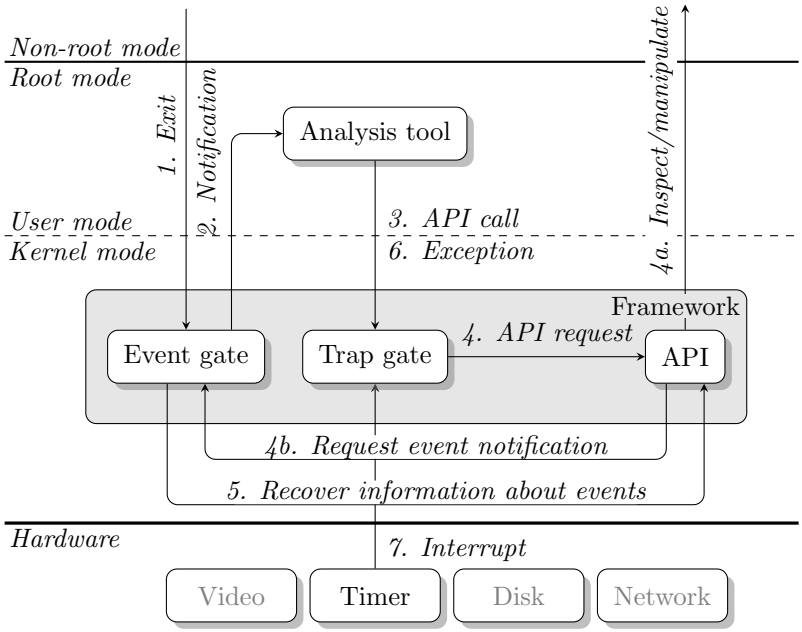

Figure 3.2: A close-up of the framework

The separation between these two parts is made possible by the fact that, when VMX is enabled, root and non-root modes offer two fully-featured execution environments. Thus, like the guest running in non-root mode, the framework running in root mode can rely on privilege separation to isolate the analysis tool and can handle independently interrupts and exceptions that might occur while executing in root mode.

When an exit to root mode interrupts the execution of the guest, the event is delivered to the *event gate* (step 1 in Figure 3.2). The event gate is responsible for abstracting low-level events into higher-level ones, and to notify the analysis tool if the latter has requested to do so (step 2). On startup the analysis tool requests the framework to be notified of certain events (not shown in the figure). The tool can use the API provided by framework to query extra information about the event (e.g., the content of the stack location storing one of the arguments of a function). Since the tool is isolated from the framework, API functions are invoked through software interrupts. Thus, requests coming from the analysis tool are received by the *trap gate* (step 3), then forwarded to the component implementing the API (step 4). The tool can perform two types of API calls: (step 4a) to inspect or manipulate the state of the guest, and (step 4b) to control event notifications (e.g., enable or disable the notification of certain events). Note that the component implementing the API is also used by the framework itself (step 5) to recover extra information about events (e.g., the return address of a function stored in the stack). The trap gate also serves the purpose of detecting exceptions (e.g., page faults) that might occur during the execution of the analysis tool. If the trap gate intercepts an exception (step 6), it terminates the faulty tool and unloads the framework, to resume the normal operation mode of the

Event	Description	Arguments
`ProcessSwitch`	Context (process) switch	—
`Exception`	Execution	Exception vector, faulty instruction, error code
`Interrupt`	Hardware or software interrupt	Interrupt vector, requesting instruction
`BreakpointHit`	Execution breakpoint	Breakpoint address
`WatchpointHit`	Watchpoint on data read/write	Watchpoint address, access type, hitting instruction
`FunctionEntry`	Function call	Function name/address, caller/return address
`FunctionExit`	Return from function	Function name/address, return address
`SyscallEntry`	System call invocation	System call number, caller/return address
`SyscallExit`	Return from system call	System call number, return address
`IOOperationPort`	I/O operation through hardware port	Port number, access type
`IOOperationMmap`	Memory-mapped I/O operation	Memory address, access type

Table 3.1: Events traceable using our framework and corresponding arguments (the argument that represents the current process is omitted, as it is common to all the events)

system. Finally, the trap gate is also used to handle timer interrupts (step 7), that, as will be discussed in Section 3.3.4, are employed to enforce a time-bound on the execution of the tool.

The functionalities provided by the API of the framework can be classified into two classes: *execution and I/O tracing* and *state inspection and manipulation*. The following paragraphs describe briefly the API. More details are given in Sections 3.3.2 and 3.3.3.

Execution and I/O tracing facilities allow a tool to intercept the occurrence in the analyzed system of certain events and certain I/O operations respectively. Table 3.1 reports the main types of events that can be traced. For each event, the table also reports the arguments associated to the event; arguments are information about the events most commonly used in tools. For example, the events `FunctionEntry` and `SyscallEntry` are used to trace functions and system calls respectively. The arguments associated to the `FunctionEntry` event are the address (or the name) of the function called, the caller and the return address. Another example is the `ProcessSwitch` event that can be used to trace context switches between processes (not threads). From the point of view of the analysis tool all the events are handled in the same way: the tool can subscribe to any event and, when the event occurs, can inspect its arguments and take the proper actions. However, at the framework-level, certain events are different from other ones. Indeed, some of them (e.g., context switches between processes) can be traced directly by the hardware. That is, the event triggering the exit corresponds exactly to the event being traced. Other events instead (e.g., function calls and returns) cannot be traced directly by the hardware. In all these cases the framework relies on other low-level events to trace the execution and then abstract exiting low-level events into higher-level ones, meaningful for the analysis tool.

Arguments can optionally be used as conditions, to limit the tracing to a sub-

set of all the events. Conditions on events serve two purposes. First, conditions allow to simplify the analysis tools, since events that do not match the requested conditions are discarded by the framework and thus do not need to be handled by the tool. Second, some conditions allow preemptive filtering of the events. In other words, the framework configures *a priori* which events trigger an exit, instead of filtering out exits caused by uninteresting events. For example, in the case of the `IOOperationPort` event, preemptive filtering means to configure the CPU such that only I/O operations involving a specific I/O port trigger an exit. This feature is very important to minimize the number of exits and thus the overall overhead.

State inspection and manipulation primitives can be used by the tool to access the state of the guest, in order to extract more detailed information about events or other data useful for the analysis. For example, these primitives allow to extract the arguments of an invoked function, or to inspect the internal structures of the guest operating system. Note that, by default, write access to guest state is not granted to a tool. If necessary, such permission can be enabled at compile-time. Obviously, in this case the framework cannot protect the state of the guest from dangerous modifications.

3.3.1 Framework and analysis tool loading

The framework and the analysis tool are loaded by a minimal kernel driver. This is unavoidable since the operations we need to perform to load the framework require maximum privileges and can be performed only by the kernel of the operating system. The driver, however, is indeed very simple and we put extreme care in avoiding any interference with the kernel. Moreover, since once loaded the framework is completely invisible to the system, we unload the driver immediately as soon as the framework has been installed.

When VMX modes are enabled, the VMCS is made accessible initially to the loader, and subsequently, when the loading is completed, only to the framework. As we mentioned in Section 3.1, the VMCS stores the *guest state* and the *host state*, i.e., the states of the processor that are loaded on entries to non-root and on exits to root mode, respectively. Furthermore, the *execution control fields* can be used to control the set of events triggering exits. The task of the loader is to enable VMX modes and to configure the VMX data structure such that the operating system and user-space applications continue to run in non-root mode, while the framework and the analysis tool are executed in root mode. In addition, the loader has to configure the CPU such that all the events necessary for the tool to trace the execution of the system trigger exits to root mode. When the initialization is completed, the driver unloads itself and resumes the execution of the system.

Guest state configuration

The guest state is initialized to the current state of the system. In this way, when the virtual machine is launched and execution enters non-root mode, the guest operating system will resume its execution as if nothing happened. A tricky problem when initializing non-root mode concerns the management of the memory. More precisely, we must prevent the newly created guest to use and access the physical memory frames allocated to the framework and to the tool. Otherwise, the guest could detect and even corrupt the framework. Most recent CPUs provide hardware facilities for memory virtualization (e.g., Intel Extended Page Table extension). If these facilities are not available, memory virtualization must be implemented entirely via software. Briefly, software memory virtualization consists of intercepting all guest operations to manipulate the page table (the data structure the CPU uses for virtual-to-physical address translation) and in ensuring that none of the physical frames allocated to the framework and to the analysis tool are mapped into the guest. In case the guest tries to map a reserved physical frame, the framework assigns the guest a different one and masquerades the difference (not available in our current implementation).

Host state configuration

The host state is initialized as follows. The CPU is configured to use, when in root mode, a dedicated address space and a dedicated interrupt descriptor table (IDT). This configuration simplifies the separation of the analyzed system from the framework and allows to detect and handle interrupts and exceptions that occur in root mode. Differently from the address of the entry point of non-root mode, which is updated at every exit to allow to resume execution of the guest from where it was interrupted, the address of the entry point of root mode is fixed. The entry point is set to the address of the routine that takes care of dispatching an exit event to the appropriate handler and that in turn might notify the analysis tool (i.e., the entry point of the event gate). Finally, we register the Global and Local Descriptor Tables (we use the same tables used in non-root mode) and we assign the stack.

Execution control fields configuration

To reduce the run-time overhead suffered by the guest system, the execution control fields are configured to minimize the number of events that trigger an exit to root mode. When the tool is initialized, it specifies which events must be intercepted. Subsequently, in response to the invocation of API functions, the configuration of the execution control fields can be altered to intercept additional events or to ignore other ones.

Event	Exit cause	Native exit
`ProcessSwitch`	Change of page table address	√
`Exception`	Exception	√
`Interrupt`	Interrupt	√
`BreakpointHit`	Debug except. / Page fault except.	
`WatchpointHit`	Page fault except.	
`FunctionEntry`	Breakpoint on function entry point	
`FunctionExit`	Breakpoint on function return address	
`SyscallEntry`	Breakpoint on syscall entry point	
`SyscallExit`	Breakpoint on syscall return address	
`IOOperationPort`	Port read/write	√
`IOOperationMmap`	Watchpoint on device memory	

Table 3.2: Techniques for tracing events

3.3.2 Execution tracing

Table 3.2 describes the techniques used to trace all the events currently supported by the framework. Low-level events (those with a mark in the last column) correspond directly to exits to root mode (e.g., `Exception`). Other events are traced through the aforementioned ones (e.g., `BreakpointHit`), and others again are traced through the latter (e.g., `FunctionEntry`).

Events that can be traced directly through the hardware are process switches, exceptions, interrupts, and port-based I/O operations. All these events exit conditionally: they exit to root mode only when requested and can have optional exit conditions to limit exits to particular situations. The remaining of this section presents how we developed the primitives for tracing higher-level events starting from the low-level ones.

Breakpoints and watchpoints are two of the most complicated events to implement. Modern CPUs provide hardware facilities to realize efficient and transparent breakpoints and watchpoints. Unfortunately, hardware-assisted breakpoints and watchpoints are limited in number (only 4) and shared between non-root and root mode. Therefore, they cannot be used simultaneously by the analyzed system and by the framework. The solution we adopt to allow an arbitrary number of breakpoints is to use *software breakpoints*. A software breakpoint is a one-byte instruction that triggers a breakpoint exception when executed. Software breakpoints are enabled by replacing the byte at the address on which we want the breakpoint with the aforementioned instruction. When the breakpoint is hit, the original byte is restored and the event is notified to the tool. If the breakpoint is not persistent the execution of the system is resumed. Otherwise the instruction is emulated and then the breakpoint is set again. Clearly, this approach to breakpoints is not transparent for the analyzed system (i.e., a malicious program could spot a software breakpoint and alter its behavior). However, it is very efficient. An alternative and transparent approach is to use the same technique we use for

watchpoints, as described in the next paragraph. Our framework supports both approaches.

The approach used in our framework to implement software watchpoints is based on protecting the memory locations from any access via hardware (or just from write accesses, depending on the type of watchpoint), such that any access results in an exception [132]. More precisely, since the finest level of protection offered by the hardware is at the page level, we mark the page containing the address on which we want to set the watchpoint as "non-present". Any future access to this page will result in a page fault exception that will be intercepted by our framework. The framework analyzes the exception and checks whether the accessed address corresponds to the address with the watchpoint. If the watchpoint is hit, the framework delivers the event to the analysis tool, otherwise it emulates the instruction, and then resumes the normal execution of the guest. Emulation is necessary to execute the faulty instruction manually. Indeed, to prevent a second fault, the original permission of the memory page accessed by the instruction must be restored before executing the faulty instruction. After the execution of the instruction, the page must be marked again as "non-present" to catch future accesses. Obviously this approach increases the run-time overhead, due to a number of synthetic page fault exceptions; however, it also guarantees a higher level of transparency to both the guest operating system and user-space applications.

Other higher-level events, such as function and system call entries and exits, are traced through breakpoints. When the analysis tool requests the framework to monitor a certain function, the framework sets a breakpoint on the address of the entry point of the function. Later, when a breakpoint is hit, the framework checks whether the hit breakpoint corresponds to a function entry point and, if so, it delivers the appropriate event (i.e., `FunctionEntry`) to the analysis tool. Function exits, instead, are traced by setting a breakpoint on the return address. The framework discovers the return address by setting a breakpoint on the function entry and by inspecting the stack frame of the function when the breakpoint on the entry point is hit. A similar approach is used for tracing system calls entries and exits.

The approach for tracing function calls and returns just described allows to trace specific functions, whose names or addresses are supplied by the tool. The tracing of all function calls and returns is instead more complicated because it is not possible to know *a priori* the addresses of all functions' entry points. The solution in this case is to perform a static analysis to identify the addresses of all functions' entry points (e.g., by recognizing function prologues). This feature is still not available in our current implementation of the framework. Nevertheless, if needed, the static analysis could be performed directly in the tool. It is worth pointing out that, due to the theoretical limitations of static approaches [64], this kind of analysis cannot always provide very accurate results, especially when dealing with highly optimized machine code. The tracing of all system calls is instead much easier, since they are all invoked through a common gate. The

solution we adopt is to put a breakpoint on the entry point of the system call gate [29].

Besides execution tracing facilities, the framework also exposes to analysis tools the possibility of intercepting I/O operations with hardware peripherals. Software can interact with hardware devices through hardware I/O ports, or it can leverage memory-mapped I/O. In the first case, VMX allows to intercept the operation without any effort: the framework simply configures the execution control fields such that all the interactions with the specific hardware ports trigger an exit to root mode; when such an exit occurs, the framework notifies the tool by means of a `IOOperationPort` event. However, for performance reasons, modern peripherals typically resort to memory-mapped I/O. In this case, read and write operations do not involve any hardware port, as they are performed directly on memory. To intercept such operations we set a watchpoint on the appropriate memory region, using the aforementioned technique. Thus, when an access to the memory region being watched is detected, the framework delivers a `IOOperationMmap` event to the tool.

3.3.3 State inspection and manipulation

Several situations require to access the state of the guest system in order to inspect, and optionally manipulate, both the registers of the CPU and the memory. As an example, the framework could need to read the return address of a function from the stack, to access the parameters of a system call from the processor registers, or to insert a breakpoint into the address space of a particular process. Similarly, the analysis tool might need to extract data from the memory of the guest.

The inspection and manipulation of CPU registers is a straightforward activity. This information is saved during an exit and restored before an entry. Thus, the inspection and manipulation of registers merely consists of reading or writing the VMX guest state (or the memory of the framework, depending on the type of register).

Inspection and manipulation of memory locations is much more complex. When paging is enabled, virtual addresses are translated by the hardware into physical addresses according to the content of the page table and direct physical addressing is not possible. Each process has its own page table; therefore, different processes have different virtual-to-physical mappings and a process cannot access the memory of the others. The framework is isolated from the guest using the same approach and thus it has its own page table and its own mapping. Consequently, the framework cannot directly access memory locations of guest processes. Moreover, inspection is complicated by the fact that page tables cannot be traversed via software (but only via hardware): the page table is a multilevel table and pointers to lower levels are physical. To overcome this problem we have developed a specific, OS-independent, algorithm to access an arbitrary virtual memory location of an arbitrary process. The core of the algo-

rithm is a primitive that allows to access arbitrary physical memory locations. This is accomplished by mapping a given physical address p to an unused virtual address v in the page table of the framework, and subsequently by accessing v. Then, using this primitive, the algorithm can traverse the page table of a process of the guest via software by iteratively mapping the physical addresses stored in the table.

The framework exposes memory inspection and manipulation facilities, based on the aforementioned algorithm, to the analysis tools through two API functions: `GuestRead(p,a,n)` and `GuestWrite(p,a,data)`. The former reads *n* bytes starting from virtual address *a* of process *p*; the latter writes the content of buffer *data* into the address space of process *p*, starting from virtual address *a*. By default, to preserve the integrity of the guest, all `GuestWrite` operations are forbidden. On top of these functions we have built higher-level ones that facilitate the extraction of functions' arguments, null terminated strings, and to disassemble code.

3.3.4 Tool isolation

To be able to use our infrastructure on a production system, it is essential to guarantee that any defect in the analysis tool will not affect the stability of the analyzed system and of the framework. At this aim, the framework controls the execution of the analysis tool and, if any anomalous behavior is observed, the whole infrastructure is automatically unloaded.

As we outlined at the beginning of this section, even if the analysis tool is executed in VMX root mode, it is still constrained into a less privileged execution mode than the framework. Thus, any operation the tool performs on the guest must be mediated by the framework. This is exactly what happens in traditional operating systems: a user-mode process cannot access directly the resources of the operating system, nor those of other user-mode processes, and any action it performs outside its address space must be mediated by the kernel. Similarly, in our context, to perform an operation on the guest system, the tool must use the programming interface offered by the framework.

In the default configuration, the framework does not allow a tool to access in write-mode to the state of the guest. However, there is still the possibility that the execution of an instruction of the tool raises an unexpected exception (e.g., a page fault on memory access, or a general protection fault). When such an event occurs, the framework has no way to handle the anomalous situation and to allow the tool to continue its execution. The only viable approach that also preserves the integrity of the guest system is to terminate the analysis tool and to remove the framework. At this aim, the solution we adopt is to intercept unexpected exceptions through the custom interrupt descriptor table (IDT) installed when launching VMX modes. The IDT receives the trap, and delivers it to the trap gate

that eventually unloads the framework[2]. Another problem that might arise with a buggy analysis tool is non-termination: if the analysis tool entered an infinite loop, the guest system would never be resumed. To prevent this problem we added to the framework a minimalistic watchdog and set a time limit on the execution of the tool. The limit is not on the whole execution time of the tool, but rather on the execution time to handle an event. Thus, the analysis tool could potentially be run forever, but with the guarantee that the execution of the analyzed system will be resumed within the specified time limit. At this aim, before delivering an event to the analysis tool, the framework resets a timer. Then, while the tool handles the event, the framework periodically regains the control of the execution and checks whether the time limit has been exceeded. To do that the framework registers, in the IDT, a custom interrupt handler to handle timer interrupts and programs the interrupt controller to deliver only timer interrupts (that is necessary to prevent the framework to consume interrupts for all the other devices). Before returning to non-root mode, the framework reprograms the interrupt controller to deliver all the interrupts to the analyzed system.

3.3.5 OS-dependent interface

Our framework provides a general programming interface completely independent from the operating system running inside the guest. However, in many cases some OS-specific facilities can ease the analysis of the guest. As an example, the only OS-independent manner to identify a process is by means of the base address of its page table (typically stored inside the `cr3` CPU register). However, it is quite awkward to refer to processes using page table base addresses, and it is more natural to identify a process through its process identifier (PID) or through the name of the application it executes.

The OS-dependent interface we provide leverages virtual machine introspection techniques [42] to analyze the internal structures of the guest operating system to translate OS-independent information (e.g., process with page table base address `0x13cdc000`) into something more user-friendly (e.g., process `notepad.exe`). Moreover, if debugging symbols are available the framework can use them to resolve symbols' names and addresses (e.g., functions and global variables). In this way, a tool can ask to interrupt the execution of the guest when function `NtCreateFile` is invoked, instead of referencing the function through its address. Similarly, when a function is invoked, it is possible to inspect its call-stack and to resolve the name of the caller functions and even to recover the libraries to which the various functions belong to. Some of the OS-dependent functionalities provided are summarized in Table 3.3.

In case the guest operating system is not supported, the OS-dependent module is disabled, and only OS-independent functionalities are available. Our current

[2]Only unexpected exceptions caused by a tool trigger the unloading of the framework. Interrupts coming from hardware devices, such as the network card or the disk, are queued and eventually injected into the guest system instead.

Name	Description
`GetFuncAddr(`*n*`)`	Return the address of the function n
`GetFuncName(`*a*`)`	Return the name of the function at address a
`GetProcName(`*p*`)`	Get the name of process with page directory base address p
`GetProcPID(`*p*`)`	Get the PID of process with page directory base address p
`GetProcLibs(`*p*`)`	Enumerate the dynamically linked libraries loaded into process p
`GetProcStack(`*p*`)`	Get the stack base for process p
`GetProcHeap(`*p*`)`	Get the heap base for process p
`GetProcList()`	Enumerate processes
`GetDriverList()`	Enumerate device drivers

Table 3.3: OS-dependent API

implementation offers an OS-dependent interface only for the Windows XP operating system.

3.4 HyperDbg

In this section we present HyperDbg, an interactive kernel debugger we built on top of our framework. The current implementation of HyperDbg supports both Microsoft Windows XP and Linux. In our strive to contribute to the open source community, we released the code of HyperDbg, along with the code of the framework, under the GPL (v3.0) license. The code is available at `http://code.google.com/p/hyperdbg/`.

HyperDbg offers all the features commonly found in kernel-level debuggers but, being completely run in VMX root mode, it is OS-independent and grants complete transparency to the guest operating system and its applications. The debugger provides a simple graphical user interface to ease the interaction with the user. This interface is activated in two circumstances: (I) when the user presses a special hot-key or (II) when the debugger receives the notification for an event that requires the attention of the user (e.g., when a breakpoint is hit). From this interface the user interacts with the debugger and can perform several operations, including setting breakpoints and watchpoints, tracing functions and system calls, and inspecting and manipulating the state of the guest (since all interactive debuggers allow to modify the state of the debuggee, we decided to enable write access to the guest as well).

Figure 3.3 shows HyperDbg in action[3]. In particular, the figure shows the debugger notifying the event that interrupted the execution of the analyzed system, displaying a fragment of the code of the process currently running in the analyzed system and displaying a "backtrace" of the function calls that are currently

[3]The screenshot was taken using our development environment based on an Intel x86 emulator supporting extensions for virtualization (i.e., BOCHS).

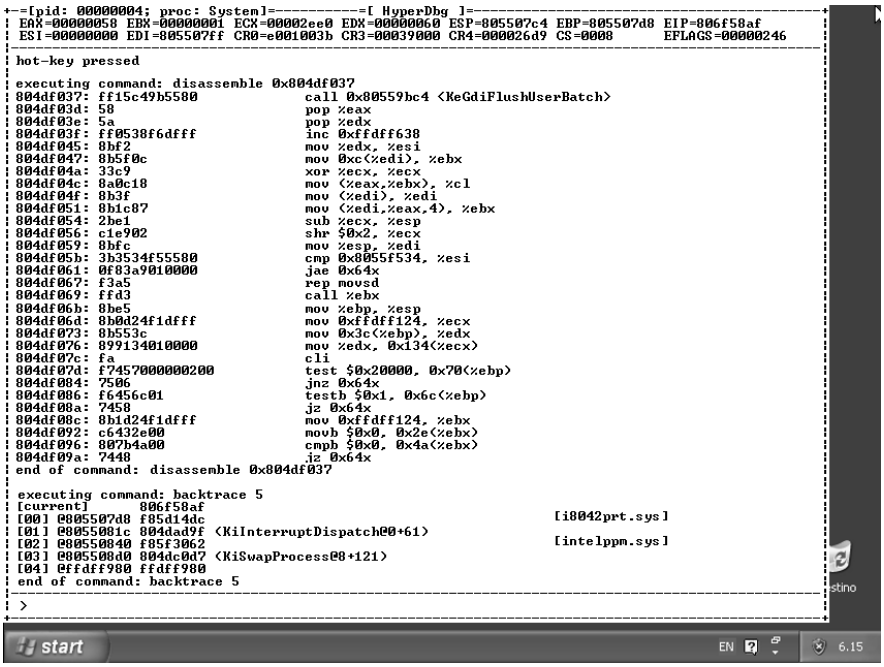

Figure 3.3: HyperDbg in action

active. Additionally, the debugger displays information about the status of the registers at the time the event occurred (in the case of the figure the event is the pressure of the hot-key). To facilitate the analysis, the debugger leverages OS-dependent information. For example, the screenshot in Figure 3.3 shows that the debugger resolved the ID and the name of the process in a Microsoft Windows XP guest, by knowing how the process table is managed by the operating system.

It is worth pointing out that HyperDbg can be used to debug *any* piece of code of the guest system, including critical components such as the process scheduler, or interrupt and exception handlers. Indeed, Figure 3.3 shows that the guest operating system has been stopped while executing the PS/2 keyboard/mouse driver (`i8042prt.sys`). Thanks to the fact that the framework on which the debugger is built on is completely transparent to the analyzed system, the user can use the keyboard to interact with the debugger even though the keyboard driver of the guest is being debugged.

HyperDbg consists of less than 1600 lines of code: ∼25% of the code implements the graphical interface, ∼23% of the code provides the facilities required for keyboard-based user interaction, and the remaining ∼52% is responsible for handling events and for all the other interactions with the framework. Note that certain functionalities (e.g., disassembling a code region) are implemented directly in the framework since, most likely, they will be used for other types of analysis as well. The framework is about four times bigger than the debugger (without considering the disassembly module embedded in the framework, as it is based on an off-the-shelf disassembler). We believe these numbers are very

significant. The number of lines of code we had to write to implement HyperDbg clearly witnesses that complex analysis tools like an interactive kernel debugger are straightforward to implement using our framework.

The remaining of this section describes how we used the facilities of the framework to implement the user interface and the component to receive commands from the user.

3.4.1 User interface

Although the graphical user interface of the debugger is rough, its implementation is very challenging. The reason of the complexity is the fact that we cannot rely on any high-level graphical facility available in the analyzed system to render the interface. Such approach would be too OS-dependent and not transparent at all. The lack of graphical primitives obliged us to interact directly with the video card. The video memory is mapped at a fixed address in the guest and thus unmodified inspection and manipulation API (i.e., `GuestRead` and `GuestWrite`) can be used by the debugger to render the interface. Note that this approach is not dependent on the OS nor on the hardware. We developed a small video library that provides basic graphical functionalities and translates our requests into data that are written directly in the memory of the video card. Before rendering the graphical interface to the screen, the debugger backups the content of the video memory and restores the content right before resuming the execution of the analyzed system.

3.4.2 User interaction

User interaction is keyboard-based. When in non-root mode, the user can switch into HyperDbg by pressing a hot-key. Then, in root mode the user can control the debugger. For these reasons, HyperDbg must be able to intercept keystrokes both in root and non-root mode. To intercept keystrokes in non-root mode we monitor all the read operations from the hardware I/O port devoted to the keyboard. In other words, HyperDbg registers to the core for all the `IOOperationPort` events that satisfy the event condition *port*=`KEYBOARD_PORT && `*access*=`read`. When such operation is detected, HyperDbg checks whether the key pressed corresponds to the hot-key that enables the debugger. If the key pressed matches the hot-key the debugger pops up the graphical interface and waits for commands. Otherwise, the debugger passes the keystroke to the analyzed system such that the latter will continue its execution as if the keystroke were read directly from the keyboard. Keyboard handling in root mode is done by polling the keyboard hardware I/O port. Since direct access to I/O ports is not permitted to any analysis tool, the debugger relies on a API function exported by the framework which mediates all accesses to I/O ports and allows (if the permission is granted at compile time) certain analysis tools to read data from certain I/O ports.

3.4.3 Real world examples

Debugging from root mode assures a number of different advantages in respect to a common ring 0 debugger. First of all advantages is transparency. There is no way, as far as we know, that code running inside the guest, regardless of its privilege level, can detect the presence of the debugger (unless the debuggee cooperates with an external entity, as we discuss in Section 3.5). This feature comes really handy when analyzing malware and root-kit code. Such pieces of code are often able to detect the presence of debuggers and other analysis tools, included virtual machine based ones [87], so that they try to deceive them to hide their presence or assume fake benign behaviors to trick the analysts.

Besides security-related aspects, HyperDbg is extremely useful for device driver developing. Indeed, writing and testing a device driver cannot be done in a virtual machine, as it would be necessary that the VM emulates the piece of hardware for which the driver is being developed and, moreover, that it does that *correctly*. Current solutions for this problem often include a secondary machine linked with a serial or USB cable to the machine where the testing is being performed [79]. Since, as we explained before, our framework can be *hot-plugged* in a running system, a driver developer just needs to load HyperDbg on the target machine and perform the debugging session *on the same machine*. To summarize, HyperDbg allows kernel-level debugging without the need of running the kernel as guest of a virtual machine or to have a debugging-dedicated second machine.

3.5 Discussion

In this chapter we described an analysis framework that provides an isolated execution environment to run security tools. Once installed, such environment cannot be altered by any malicious program. However, if the system to be analyzed has already been infected, then a malware controlling the machine can still tamper with the framework *before* it is installed, thus making the whole infrastructure completely ineffective. In this situation, a possible solution is to *attest* that the module responsible for loading the analysis infrastructure runs untampered, and that the framework itself is not modified by the malware during the installation process. We address these problems in Chapter 4.

The effectiveness of the VMM-based analysis framework we propose depends on the impossibility to detect its presence from the guest. There has been much discussion about the transparency of hardware-assisted hypervisors: several researchers believe hypervisors necessarily introduce some discrepancies and suggest a number of detection strategies [34, 1], while others insist this is not always the case, and that similar techniques can be evaded [104]. In our opinion, writing an undetectable hypervisor is extremely challenging, if not unfeasible [40]. For example, timing analysis is very effective in detecting running VMMs, especially when the analysis is performed by an external entity, with a real perception of time. Indeed, in Chapter 4 we propose a software-based attestation primitive

that leverages this technique to detect unwanted software running in root mode. The reader could argue that malware authors can use similar strategies also to detect our VMM-based analysis framework. Anyway, we believe it is not realistic to assume that a malicious program can rely on an external entity to perform the timing analysis, and internal time sources can be altered by the hypervisor. Alternatively, malware might attempt to detect our running hypervisor by trying to install another VMM. One approach to contrast such attempts is to let the malware believe that virtualization support is not available at all.

Secondly, it is worth noting that hypervisor detection techniques can check if a VMM is installed on the system, but they cannot discern what kind of hypervisor is actually present. With the widespread diffusion of virtualization technologies, this kind of detection is often too coarse grained. A malware that hides its malicious behavior on a machine only because it found a running hypervisor will not be able to infect many of the virtualized environments in use today.

To conclude, the current implementation of our framework needs to be further improved in order to provide a reliable analysis platform. Most notably, our prototype lacks a robust interrupt handling module. As we briefly mentioned before, when executing in root mode device interrupts directed to the monitored system should be intercepted by the hypervisor, queued and injected into the guest later. The strategy we use in our current prototype is to simply ignore interrupts coming from the hardware peripherals: the hypervisor does not send to the interrupt controller any acknowledgement, so interrupts are periodically retransmitted and eventually acknowledged by the guest system. This solution is quite rough but it works pretty well in practice, even when the CPU executes in root mode for several minutes. However, in more complex scenarios this technique cannot be adopted. As example, to support user interaction through USB keyboards (now a *de facto* standard on desktop computers), HyperDbg must be able to handle interrupts coming from the USB controller, and there is no easy way to enable only USB keyboard interrupts and to disable all the other ones. We are working to address this and other implementation issues. To this end, we decided to release our analysis infrastructure and HyperDbg under the GPL license, in the hope of stimulating a broader involvement of the open-source community.

4

Software-based code attestation

Proactive detection of malicious code is not always possible. Despite the continuous advances in anti-malware technology, it is often the case that some nasty malware bypasses security protections and eventually infects a system. In these cases, the only viable alternatives are *a posteriori* approaches: it is necessary to run a program on the infected system to eradicate the malware and remediate all the effects of the infection. But how to guarantee that the malicious program that runs on the compromised machine cannot tamper with the execution of the remediation software?

A solution to this problem is to *attest* the execution of the anti-malware tool. More precisely, *code attestation* is the process of verifying the integrity of a piece of code executing in an untrusted system. Besides integrity verification, code attestation can also be used to execute an arbitrary piece of code in an untrusted system with the guarantee that the code is run unmodified and in an untampered execution environment. In the last years, hardware extensions, such as TPM chips [47], have been proposed for securing computations, including performing attestation. However, these extensions are not yet available on every computing device. In such a situation, pure software-based solutions are the only possibility.

Several different software-based attestation schemes have been proposed in literature [39, 112, 109, 108, 111, 113]. All these schemes are based on a challenge-response protocol involving two parties: an *untrusted system* and a *verifier*. The verifier issues a challenge for the untrusted system, where the challenge consists in computing the checksum of certain memory locations and properties of the execution environment. The checksum is computed by executing a particular *attestation routine*, or *checksum function*. Once computed, the checksum is sent back to the verifier. The verifier relies on the time to determine whether the checksum is genuine or if it could have been forged. Indeed, attestation routines are constructed such that any tampering attempt results in a noticeable increase of the execution time. Thus, a checksum received too late is a symptom of an attack.

The complexity of the attestation routine depends on the hardware character-

istics of the untrusted system on which it has to be executed. Indeed, the output of the routine is guaranteed to be genuine only if it is executed in a properly configured execution environment. In complex hardware architectures, such as the ones used in personal computers, there exist several configurations of the execution environment that can be exploited by an attacker to thwart attestation. Therefore, the attestation routine must ensure, and prove to the verifier, that the execution environment in which it executes satisfies all the requirements to impede attacks. In other words, the attestation routine must attest its own code, but also the execution environment. Intuitively, the requirements for tamper-proof attestation are that the attestation routine must be executed at the highest level of privilege (i.e., at the same level of the most powerful attacker) and that its execution must be uninterruptible. Practically speaking, in a legacy system with no hardware support for virtualization, that means that the routine must execute in system mode (i.e., the privilege level of the operating system) and that all interrupts must be disabled, to prevent the attacker to regain the control of the execution at some point. Unfortunately, even if the requirements are very well defined, guaranteeing that they are satisfied in a complex execution environment where attacker and defender have the same privileges is a very challenging problem.

In this chapter we present Conqueror, a software-based scheme for tamper-proof code execution on untrusted legacy systems. Conqueror provides a security primitive that allows to build applications that require the availability of a trusted computing base. Pragmatically speaking, Conqueror guarantees that an arbitrary piece of code can be executed untampered in an untrusted system, even in the presence of malicious software. Conqueror has been developed to address the limitations of Pioneer, the state-of-the-art software-based attestation solution [111]: Conqueror is immune to all attacks that are known to defeat Pioneer, and it can also be used on untrusted systems where the attacker could leverage hardware virtualization extensions to hold control of the execution environment in which the attestation routine executes. Conqueror adopts a variation of the challenge-response protocol used in traditional attestation schemes: the challenge does not consist in a seed to initialize a constant attestation routine, but instead consists in an entire routine, that is different each time, self-decrypting, and obfuscated. The intent is to make it impossible for an attacker to reverse engineer the logic of the checksum computation, and to facilitate the hiding of the sensitive operations that Conqueror needs to perform to attest that the state of the environment executing the code impedes any attack. The strength of this approach is that we are drastically increasing the time needed by an attacker to forge a checksum.

We experimentally demonstrate our claims about Conqueror's resistance to attacks. We show that even a preliminary low-level analysis of the code of Conqueror's one-time attestation routine (i.e., disassembly), which is necessary to perform any subsequent meaningful analysis for reconstructing the semantics, costs about the same time required to execute the routine. Moreover, we show that Conqueror is also resilient to dynamic attacks performed by an attacker leveraging

a hardware-assisted hypervisor. Finally, to demonstrate Conqueror's potential, we present a proof-of-concept software-based primitive to launch securely a hypervisor in a running untrusted system, to segregate the system into a restricted guest. This primitive could be used in place of `skinit` [2] and `senter` [47] on untrusted systems with no hardware support for trusted computing.

4.1 State-of-the-art of attestation on legacy systems

This section presents Pioneer, Conqueror's main competitor. Both systems target the same hardware architecture, but they use very different approaches. Moreover, Conqueror is resistant to attacks that are known to defeat Pioneer.

Pioneer is a software-based attestation scheme that can be used to establish a trusted computing base, called *dynamic root of trust*, on an untrusted legacy system. Pioneer is specific for Intel x86 with EM64T extensions. The code of the dynamic root of trust is guaranteed to be unmodified and to execute in a *tamper-proof execution environment*. The dynamic root of trust measures the integrity of an arbitrary executable, and then runs the executable in the trusted execution environment. The dynamic root of trust is established using a *verification function*. The verification function is an extension of a conventional checksum function and additionally includes a hash function to verify the integrity of an executable. The verification function is self-checking (i.e., it attests its own code), and it attests the execution environment.

The Pioneer verification function is composed of three components: (I) a checksum function, (II) a send function, and (III) a hash function. The checksum function is used to compute a checksum over the entire verification function and to setup the execution environment in which the other functions are guaranteed to run untampered. Since the sensitive component of Pioneer is the checksum function, we do not overview the others.

As in the majority of code-attestation schemes, in Pioneer the checksum function is known a priori and the challenge issued by the verifier consists in a seed that initializes this function. Therefore, an attacker has complete access to the checksum function and can analyze it offline to find weaknesses. The checksum function has been constructed manually to be time-optimal: no adversary function that can compute the correct checksum without introducing a noticeable overhead exists. Time-optimality is achieved using operations that prevent parallelization, that have a low variance execution time, and by executing these operations iteratively, to maximize the overhead of the attacker. Most importantly, the checksum function is responsible for initializing the execution environment and for attesting the correct initialization.

Unfortunately, since the hardware architecture for which Pioneer was developed is full of subtle details, researchers have found ways to thwart the setup of the dynamic root of trust without being noticed by the verifier. For example,

it is possible to perform the entire checksum computation in user-space and to regain the control of the execution through exceptions without corrupting the checksum. Another attack consists in desynchronizing data and code pointers and to execute a modified checksum function that computes the checksum of a pristine function residing elsewhere in memory [139]. Finally, Pioneer's assumptions that the most powerful attacker operates in system mode does not hold on new commodity hardware with support for virtualization [2, 53].

The remaining of this section briefly presents some attacks against Pioneer.

Checksum computation in user mode. One of the requirements to create a tamper-proof execution environment is that the verification function must operate at system level. Pioneer verifies that such requirement is satisfied by checking if the verification function can disable maskable interrupts. However, it is possible to execute the checksum in user mode, and to simulate that maskable interrupts have been disabled.

Dedicated stack for interrupt handling. Pioneer disables interrupts to prevent an attacker to interrupt the execution of the checksum function. However, not all interrupts can be disabled (e.g., exceptions). The solution adopted by Pioneer is to store part of the checksum in the stack. If an exception handler is executed to respond to an interrupt, the checksum will get corrupted because the CPU saves its state on the stack before invoking the handler. An attacker can execute his malicious interrupt handlers without corrupting the checksum by configuring the CPU to use a dedicate stack for interrupt handling.

Segments and TLB desynchronization. An attacker can modify the code of the checksum function to compute the checksum over a pristine copy of the memory. Pioneer adopts several tricks to defeat this kind of attack. Nevertheless, it is vulnerable to variations of this attack performed by desynchronizing code and data segments or desynchronizing code and data TLBs [139, 122].

Attestation in the presence of a hypervisor. Pioneer assumes that the most powerful attacker operates in system mode. This assumption can be violated on new commodity hardware with support for virtualization [2, 53], where the most powerful attacker operates in *hypervisor mode* (or root mode, according to Intel terminology) and is completely transparent to guests, Pioneer included.

4.2 Conqueror overview

In this section we give an overview of Conqueror, our scheme for software-based code attestation and tamper-proof code execution on untrusted legacy systems (Intel x86). Conqueror does not suffer the problems that affect the state-of-the-art attestation scheme for this class of systems.

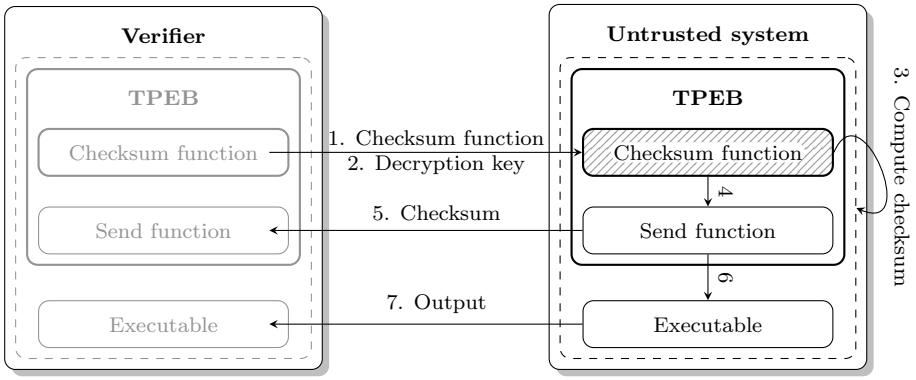

Figure 4.1: Overview of Conqueror

4.2.1 Threat model

Conqueror has been developed to operate in the following adversary scenario. We assume that the untrusted system has been compromised, and that the attacker operates at the highest privilege level: system mode (ring 0) if the system has no support for hardware-based virtualization, hypervisor mode if the support is available. However, we assume the adversary cannot operate in system management mode, that he cannot perform hardware-based attacks (e.g., DMA-based attacks or overclocking), and that he cannot leverage a pristine or a more powerful system to break the attestation scheme. The final assumption is that the untrusted system supports a single thread of execution (e.g., no SMP).

4.2.2 Conqueror architecture and protocol

As any other software-based code attestation scheme, Conqueror is based on a challenge-response protocol, where a verifier challenges the untrusted system. The central component of Conqueror is the Tamper-Proof Environment Bootstrapper (TPEB). As the name says, the TPEB is responsible for setting up the environment in the untrusted system for the tamper-proof execution of an arbitrary executable. Figure 4.1 shows the layout and the protocol of Conqueror (the numbers in the figure represent the temporal ordering of the events). The TPEB is composed of a checksum function and a send function. The checksum function computes the checksum to attest the integrity of the TPEB itself and the integrity of the executable. The send function transmits the computed checksum value to the verifier and invokes the executable. The send function is logically separated from the checksum function because it is hardware dependent (i.e., it depends on the network card installed on the untrusted system).

In Conqueror the verifier generates the checksum function on demand, such that each function differs considerably from the others. Differences are both syntactic and semantic. Moreover, functions are obfuscated using multiple obfuscation schemes. The attacker has no access to the checksum function ahead of time and cannot perform any offline analysis nor optimization [113]. In Conqueror, the

newly generated checksum function is initially sent encrypted to the untrusted system. Later on, at time t_0, the verifier transmits the key for decryption. Since the verifier knows precisely in which execution environment the function must be executed and knows the hardware characteristics of the untrusted system, it can compute the expected checksum value and can estimate the amount of time that will be required by the untrusted system to decrypt, to execute the function, and to send back the result. Let $t_1 = t_0 + \Delta_t$ be the time by which the correct checksum has to be received by the verifier to be considered authentic; Δ_t is an upper bound, empirically estimated, of the maximum time requested by the untrusted system to compute the checksum in the absence of an attack (including network delay). If the verifier does not receive the correct checksum by t_1, then the checksum is considered forged and the execution environment not tamper-proof. In a traditional checksum function (e.g., that used in Pioneer), where the function is known a priori and can be analyzed offline, the attacker has Δ_t time to execute a malicious function to forge the checksum. In Conqueror, the attacker has Δ_t to (I) analyze the checksum function, (II) generate a new function capable of forging the checksum, and (III) execute the generated function. Alternatively, the attacker would have to emulate the entire execution of the checksum function. Differently from traditional checksum functions, the ones in Conqueror are generated automatically; for this reason we cannot guarantee a low collision rate nor that their implementation is optimal (in terms of execution time and in code size). Nevertheless, given the small time frame available, there is no opportunity for the attacker to reverse engineer their semantics, nor to emulate the execution, and to forge checksums in time.

Since Conqueror targets a very complex hardware architecture, particular attention has to be devoted to prevent checksum forgery, by tampering either the checksum function or the execution environment. To attest the trustworthiness of the environment, the verifier embeds in the checksum function several operations whose behavior and execution time depend on the configuration of the environment (e.g., instructions that raise exceptions when executed without enough privileges).

An attacker who tampers the execution of the checksum function will corrupt the checksum, or will incur in a time overhead that will cause the overall checksum computation to exceed the expected time Δ_t. For these reasons, Conqueror guarantees that a correct checksum, received by the verifier by t_1, is the proof that the checksum function has been executed unmodified and that the bootstrap of the tamper-proof execution environment succeeded.

4.3 Conqueror implementation

Conqueror current implementation is specific for the Intel x86 architecture and so are the details of the implementation presented in this section. However, we believe the same scheme can be used, as is, on the Intel x86-64 architecture.

4.3.1 Tamper-Proof Environment Bootstrapper

The layout in memory of the TPEB is shown in Figure 4.2. The TPEB consists of the checksum function, its data, and the send function. For simplicity, the TPEB is located at a fixed address (`BASE`) and in consecutive memory pages. Moreover, the executable follows immediately the TPEB, and the overall buffer is padded to a multiple of page size (`SIZE`). We assume that the TPEB is already initialized on the untrusted system, with the exception of the checksum function. The checksum function and its data reside in a dedicated memory page (starting from `BASE`) and all unused bytes in this page are initialized randomly, to hide code and data. This page is generated on-demand by the verifier and transmitted encrypted to the untrusted system. The latter stores in memory, at the `BASE` address, the page and waits for the decryption key. Attestation begins when the verifier sends out the key. The reason for encryption is to exclude from the measurement the time required to transmit and prepare the TPEB.

4.3.2 Checksum function

The checksum function is composed of a prologue, a checksum loop, and an epilogue (Figure 4.2). The prologue decrypts the rest of the page containing the checksum function, initializes the execution environment for the remaining of the computation, and invokes the checksum loop. The checksum loop (described in Section 4.3.2) computes the checksum of the memory pages containing its own code, the send function, and the module we want to execute (i.e., from `BASE` to `BASE + SIZE`), and invokes the epilogue. The epilogue invokes the send function, which in turn invokes the executable module.

The checksum function computes the checksum by combining multiple checksum gadgets. In the current implementation the checksum size is 128 bits. A *gadget* (c_i) is a small code snippet that receives in input the address of a memory location and updates the running value of the checksum, according to the content of the memory. We refer to these gadgets as *active*, since they are intentionally executed by the checksum function. The purpose of an active gadget is twofold. First, each gadget contributes to the computation of the checksum in a different way. Thus, the correct checksum can be computed only if all the gadgets are executed in the proper order and with the proper arguments. Second, certain gadgets perform additional operations to verify the trustworthiness of the execution environment and, in case the environment has been tampered, they either corrupt the checksum or introduce a time overhead. Since gadgets are scattered around the memory, differ syntactically and semantically from one checksum function to another, and are obfuscated, it becomes very difficult for the attacker to reconstruct the exact logic of the checksum function.

In addition to active gadgets, the checksum function relays on *passive gadgets* (h_j), or *handlers*, that are not invoked directly by the checksum function, but rather as the result of an unexpected event that can occur only in a tampered ex-

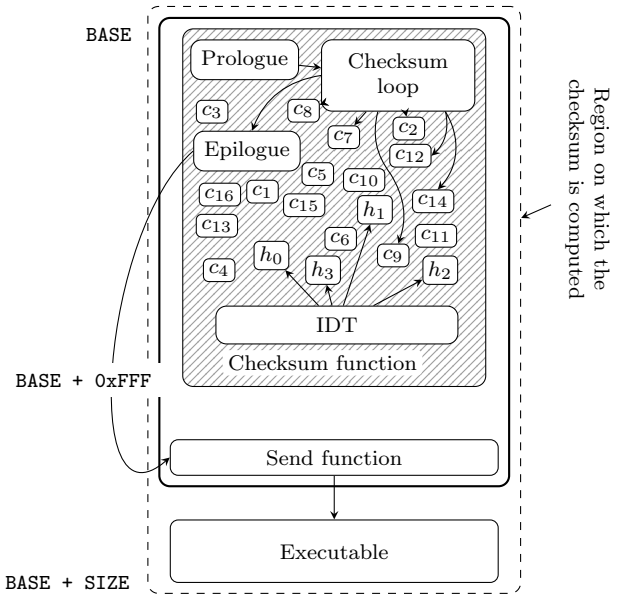

Figure 4.2: Overview of the TPEB

ecution environment. If executed, passive gadgets corrupt the checksum. Passive gadgets are registered during the prologue, by replacing the Interrupt Descriptor Table (IDT) with a new one embedded within the TPEB, and cannot be disabled by the attacker: an improper configuration of these gadgets will result in a wrong checksum.

Prologue

The prologue (Figure 4.3) is a small routine that decrypts the rest of the page and initializes the trusted execution environment. More precisely, the prologue disables all maskable interrupts (line 2), decrypts the rest of the page (line 4 and 5), and installs custom interrupts handlers (line 7). Custom handlers are installed by updating the address of the interrupt descriptor table (IDT). The new address is set to a location, within the memory page containing the checksum function, that holds a pre-initialized IDT (Figure 4.2). The mapping between interrupts and handlers (the content of the IDT) is chosen by the verifier and not known to the attacker. The handlers (h_i in Figure 4.2), or passive gadgets, are a special type of gadget: like normal gadgets they modify the running value of the checksum, but they terminate their execution with a special instruction to return to normal execution (i.e., `iret`). Furthermore, handlers are never invoked explicitly by the checksum loop but only in response to interrupts or exceptions.

The purpose of the prologue is twofold. First, by disabling maskable interrupts (pin-based interrupts generated by the peripherals) we inhibit the asynchronous execution of all handlers. Second, by installing custom interrupt handlers that update the checksum value, we can tell whether any interrupt or exception occurred during the computation of the checksum. If maskable interrupts are successfully

disabled, no asynchronous interrupt occurs, and the checksum is not corrupted because no interrupt handler is fired. Similarly, if the checksum loop executes privileged instructions, and the checksum function is executed in system mode, no exception occurs and no exception handler corrupts the checksum. On the other hand, any attempt to execute the checksum function in user mode results in an exception, in the execution of the corresponding handler, and in a corruption of the checksum value.

By positioning the IDT in the same memory page of the checksum function, we implicitly certify the content of the table. The only opportunity for the attacker is to intercept and simulate a successful update of the IDT. For example, the attacker could emulate the execution of the prologue or execute the prologue in user-space, such that the update of the IDT will raise an exception and will be intercepted. Then, the attacker could install his own malicious IDT and simulate a successful disabling of maskable interrupts. We prevent this attack by including in the checksum loop a special gadget that queries the address of the IDT and updates the running value of the checksum accordingly. Therefore, attacker's attempts to relocate the IDT will result in a corrupted checksum. Further details about the aforementioned gadget and about why its execution cannot be detected by the attacker are given in Section 4.3.2.

In conclusion, a correct value of the checksum, received by the verifiers within the expected time, certifies that the prologue is executed successfully, that the checksum function is executed at the maximum privilege level, and that the attacker cannot interrupt the execution using interrupts or exceptions.

Checksum loop

The core of the checksum computation is the checksum loop shown in Figure 4.4. The checksum loop is composed of two nested loops. The innermost loop traverses the memory and updates the checksum according to the content of the memory, invoking a different gadget at each iteration. The memory is not traversed linearly, but instead in a pseudorandom fashion (line 4), using a T-function [61]. The T-function produces a pseudorandom permutation of all the memory locations to traverse. More precisely, the T-function returns the memory offset of the next memory location for the checksum computation. At each iteration (line 5), from the offset returned by the T-function, the checksum loop computes the absolute address of the memory location to process, and invokes a specific gadget to update the running value of the checksum (`GADGETS` represents the number of gadgets available). Clearly, without an analysis of the code, the attacker cannot predict which gadgets will process which memory locations and, even if the checksum function were weak (e.g., it suffers a high collision rate), the attacker would not have enough time to exploit the weakness. Finally, it should be noted that the execution of the checksum loop is deterministic, unless it is tampered.

The outermost loop repeats the memory traversal multiple times (`ITERATIONS` denote the number of iterations of the outermost loop). At each iteration, the

```
1 // Disable maskable interrupts
2 asm("cli");
3 // Decrypt the remaining of the page
4 for (i = PROLOGUE_SIZE; i < 4096; i++)
5   BASE[i] ^= KEY[i % KEY_SIZE]
6 // Install custom interrupt handlers
7 asm("lidt %0" : : "m" (IDT));
```

Figure 4.3: Overview of the prologue

```
1 for (i = 0, j = 0; i < ITERATIONS; i++) {
2   x = seed(i) % (SIZE / 4);
3   do {
4     x = (x + (x*x | 5)) % (SIZE / 4);
5     checksum_gadget[j++ % GADGETS](BASE + x*4);
6   } while (x != seed(i) % (SIZE / 4));
7 }
```

Figure 4.4: Overview of the checksum loop (in C for clarity)

T-function used in the innermost loop is initialized with a different seed (line 2). Therefore, the innermost loop is executed multiple times and at each execution the running value of the checksum is updated using a different combination of memory locations and gadgets, and the order in which the checksum is updated is also different. Since the checksum function is constructed such that any attacker's attempt to forge the correct checksum will introduce an overhead in the computation of the checksum, the outermost loop causes a constant time overhead per iteration and facilitates the detection of the attack. Details about how we select the optimal number of iterations for the outermost loop are given in Section 4.4.

The seeds used by the T-function to generate the addresses are also included in the memory page containing the checksum function. To avoid wasting precious bytes of the page, the vector containing the seeds is positioned at a random location within the page and is not initialized, to overlap with the existing content of the page.

Checksum gadgets

The checksum is computed by executing a sequence of gadgets, each of which contributes to update the running value of the checksum in a different way. Certain gadgets also perform additional operations to attest the trustworthiness of the execution environment. Given that gadgets are very small in size and that an entire memory page is dedicated to the checksum function, the checksum function can rely on about a hundred different gadgets simultaneously. Gadgets are generated on demand by the verifier and change (in number, position, syntax, and semantics) from challenge to challenge.

The following paragraphs describe in details the gadgets used in the checksum function to attest the integrity of the TPEB and of the code of the executable. Figure 4.5 shows some sample gadgets. For clarity, the gadgets presented are not optimized and use symbolic names (in uppercase) to refer to absolute memory locations containing data: `CHKSUM` and `ADDR` refer respectively to the memory locations storing the 128-bit checksum and the address of the next word to process.

Plain checksum computation. The simplest and most frequently used gadget is responsible only for updating the running value of the checksum. Different

```
1 mov ADDR, %eax
2 mov (%eax), %eax
3 xor $0xa23bd430, %eax
4 add %eax, CHKSUM+4
```

(a)

```
1 mov ADDR, %eax
2 mov (%eax), %eax
3 add %eax, CHKSUM+8
4 sidt IDTR
5 mov IDTR+2, %eax
6 xor $0x6127f1, %eax
7 add %eax, CHKSUM+8
```

(b)

```
1 mov ADDR, %eax
2 mov (%eax), %eax
3 xor $0x1231d22, %eax
4 mov %eax, %dr3
5 mov %dr3, %ebx
6 add %ebx, CHKSUM
```

(c)

```
1   mov ADDR, %eax
2   mov (%eax), %eax
3   lea l_smc, %ebx
4   roll $0x2, 0x1(%ebx)
5 l_smc:
6   xor $0xdeadbeef, %eax
7   add %eax, CHKSUM+4
```

(d)

```
1 mov ADDR, %eax
2 mov (%eax), %ebx
3 and $0xfffff000, %eax
4 add $0x2b8, %eax
5 movb (%eax), %cl
6 movb $0xc3, (%eax)
7 call %eax
8 movb %cl, (%eax)
9 xor $0x7b2a63ef, %ebx
10 sub %ebx, CHKSUM+8
```

(e)

```
1 mov ADDR, %eax
2 mov (%eax), %ebx
3 vmlaunch
4 xor $0x7b2a63ef, %ebx
5 sub %ebx, CHKSUM+8
```

(f)

Figure 4.5: Sample gadgets for (a) plain checksum computation, (b) IDT attestation, (c) system mode attestation, (d,e) instruction and data pointers attestation, and (f) hypervisor detection.

gadgets update the checksum in different ways, by applying different arithmetical or logical operations and by modifying different bits of the checksum value. Figure 4.5(a) shows a sample gadget. The gadget updates the checksum by adding the result of a bitwise XOR between the current memory location (`ADDR`) and a random key (`0xa23bd430`). Note that this gadget modifies the second word of the running 128-bit checksum (`CHKSUM+4`, at line 4).

IDT attestation. During the prologue, the interrupt descriptor table is replaced with a custom table, which is provided along with the checksum function. Since the prologue is executed at the beginning of the checksum function, it is reasonable to expect the attacker to try to emulate or intercept its execution.

The content of the IDT is implicitly attested by the normal checksum computation, but the address of the IDT is not. To attest that the IDT shipped with the checksum function is actually being used, the checksum function relies on a specific gadget that queries the CPU to obtain the address of the IDT and updates the checksum accordingly. Obviously, the checksum will be wrong if a different IDT is being used. The only opportunity for the attacker to force the checksum function to behave as if the requested IDT were successfully installed is to intercept the query and to manipulate its output. To query the address of the IDT, the gadget uses the `sidt` instruction. Unfortunately for the attacker, this instruction is not privileged: it does not trigger an exception when executed in user mode [100]. Consequently, the only solution for the attacker to detect the instruction is to analyze the checksum function or to emulate its execution. However, any analysis or emulation attempt will introduce a noticeable overhead

in the computation of the checksum. Figure 4.5(b) shows a sample gadget to attest the IDT. The only difference with a plain gadget (Figure 4.5(a)) is the addition of the instructions to query the address of the IDT (lines 4 and 5).

System mode attestation. After the update of the IDT, the attacker cannot regain the control of the execution, because all interrupts and exceptions will be served by the handlers installed by the checksum function. Although the previously described gadget forces the attacker to install our IDT, he could still attempt to execute the entire checksum function in user mode. If no maskable interrupt occurred during the execution of the checksum function, the checksum would not get corrupted, and the attack would not be detected. However, even if we suppose that the attacker executed the checksum function in user mode and that he were able to reprogram the interrupt controller to prevent any interrupt, he would lose any opportunity to regain the control of the system after checksum computation.

To have the guarantee that the TPEB is operating in system mode, the checksum function relies on a specific class of gadgets. These gadgets use a privileged instruction to update the running value of the checksum. If the function is executed in system mode, all the instructions of the gadgets will be executed successfully. However, if the function is executed in user mode, the privileged instruction will raise an exception (because of the lack of privileges), and the exception handler we installed to handle the exception will corrupt the checksum. In some cases, the handler could also trigger an endless loop. An example of such a gadget is shown in Figure 4.5(c). The gadget uses the CPU register `dr3` to store an intermediate result during the computation of the new checksum value. This register can be accessed only in system mode and any access originating from user mode causes a general protection fault exception.

Instruction and data pointers attestation. The checksum function is a self-checksumming function. A common class of attacks against self-checksumming functions are *memory copy attacks*, that allow attackers to forge checksums [111]. Briefly, in a memory copy attack, the attacker modifies the instructions of the checksum function, or the execution environment, to redirect all memory reads to memory locations containing a pristine copy of the data to attest. A memory copy attack can be performed in different ways: (I) by patching the instructions of the checksum function to read from different locations, (II) by configuring segmentation to separate the code from the data segment, and (III) by desynchronizing the data and the instruction TLBs [139].

To prevent memory copy attacks, the checksum function uses a specific type of gadget that guarantees that reads, writes, and fetches involving the same virtual memory location refer to the same physical location. Indeed, data and instruction physical pointers equivalence is sufficient to guarantee that no memory copy attacks of type (II) and (III) can be performed. We intentionally do not consider the case of memory copy attacks of type (I), performed by patching or

by emulating the checksum function, because of the noticeable time overhead the attacker would suffer. To validate the equivalence of data and instruction pointers we leverage a gadget based on self-modifying code [43]. The gadget updates the running value of the checksum by performing an operation that is generated dynamically by modifying the code of the checksum function in place. If no memory copy attack is being performed, the data pointer (used for both reads and writes) and the instruction pointer point to the same physical page. Thus, the memory write executed by the gadget to update its instruction modifies the physical page that is also being executed. If the attacker were performing a memory copy attack, the data and the instruction pointer would point to two different physical pages and the instruction executed to update the checksum would differ from the ones just created by the gadget. Consequently, the out-of-date instruction would corrupt the checksum.

Figure 4.5(d) shows a sample gadget used by Conqueror to prevent memory copy attacks. The gadget updates the checksum by adding the data read from the memory (lines 1, 2, and 7). Before the addition, the word read is XORed with an immediate (line 6). The immediate is rotated (by two bits) at each execution of the gadget, by modifying the operand of the instruction in place (line 3 and 4). In the case of a memory copy attack the checksum would not be updated correctly because the operand of the `xor` instruction would remain unmodified.

Note that, in the case of a memory copy attack of type (III), the attacker can operate on each page separately. The aforementioned gadget successfully protects against the desynchronization of data and instruction pointers that point to the page containing the checksum function, but, as is, it is ineffective at protecting other pages (containing the send function and the executable). Indeed, only instructions residing in the page containing the checksum function are executed during the checksum computation. To address this problem, we use a variation of the original gadget, that places a temporary small snippet of code (e.g., a `ret` instruction) in a random position of the input page, executes the snippet, and restores the original content of the modified locations. Figure 4.5(e) shows an example of this type of gadget. The gadget selects a random location in the page being attested (lines 1 to 4), saves the content of the location (line 5), replaces the content with a `ret` instruction (line 6), executes the newly generated instruction (line 7), restores the original content of the modified location (line 8), and finally updates the checksum (line 9 and 10).

Hypervisor detection. An attacker operating in hypervisor mode, on a system with hardware support for virtualization, has complete control of the operating system: he can intercept the execution of all sensitive instructions, interrupts, exceptions, and, most importantly, the hypervisor and the attacker are completely transparent to guests. Dai Zovi and Rutkowska *et al.* have clearly demonstrated what an attacker can do on systems with hardware support for virtualization [27, 103]. The gadgets presented so far are effective at attesting the trustworthiness of the execution environment only if we can guarantee that no

attacker can operate in hypervisor mode. Therefore, the checksum function that attests the existence of a tamper-proof execution environment on the untrusted system must be adapted to compute the correct checksum value, in the expected amount of time, only when no hypervisor is running on the system.

There is a rich ongoing debate among researchers about hypervisors detection and hiding. Although the hardware has been specifically designed to masquerade the existence of a piece of code running in hypervisor mode, everybody has become aware that constructing a completely transparent hypervisor is fundamentally infeasible and impractical from a computational and engineering perspective [40]. Indeed, hypervisors introduce several discrepancies, especially in terms of resources and timings. Our goal is to exploit these discrepancies, in particular timing discrepancies, to detect when the execution environment could not guarantee untampered execution. The main advantage we have over attackers is that checksum validation is performed by an external party, the verifier, that has a real perception of time. We exploit this advantage by including in the checksum function special gadgets that execute instructions that unconditionally trap to the hypervisor. Similarly to exceptions, hypervisor traps cause the CPU to spend several cycles to transition from system (or user) mode to hypervisor mode, to execute the handler of the hypervisor, and to transition back to system mode. By periodically executing such instructions, we cause a noticeable time overhead when a hypervisor is running on the untrusted system.

Figure 4.5(f) shows a sample gadget we use to detect hypervisors. The gadget reads a word from the memory (line 1), executes a `vmlaunch` instruction (line 3), and then updates the checksum (line 4 and 5). Other instructions, such as `cpuid`, `vmread`, and `vmcall`, can be used for this purpose. The `vmlaunch` instruction is available only on CPUs with hardware support for virtualization. Furthermore, the instruction can be executed only when virtualization support has been enabled. If a hypervisor is running on the untrusted system, any attempt to execute the instruction results in a trap to the hypervisor. In any other situation the CPU refuses to execute the instruction and generates an illegal operation exception. Recall that, by installing a custom IDT, we register handlers for all exceptions and that these handlers modify the running value of the checksum. In particular, the handler for the illegal instruction exception we install additionally updates the address of the faulty instruction for resuming the normal execution of the checksum function from the next one. That is necessary to prevent an endless loop. To not interfere with the correct checksum computation, after the trap, the attacker has to reproduce the situation that would occur on a system without hypervisor: he has to inject an illegal instruction exception into the guest to trigger the handler registered during the prologue. If the attacker mimics exactly the behavior of the CPU in the absence of the hypervisor, the checksum is computed correctly. However, the cost of the trap, of the execution of the logic to handle the trap, of the event injection, and of the exception handling we have on a system controlled by an attacker operating in hypervisor mode is much higher than the cost of the mere exception handling that we would have on a system without

hypervisor. In conclusion, the gadget takes much longer to execute in an insecure execution environment. By executing this type of gadgets multiple times during the checksum loop we have the guarantee that, if the checksum computation produces the correct return value and it does not exceed the expected computation time, the execution environment is tamper-proof.

It is worth noting that if the attacker attempted to execute the checksum function directly in hypervisor mode, he would never be able to regain the control of the execution (this is the same case of an attacker that executes the checksum function in system mode without any hypervisor).

4.3.3 Obfuscation

After generation, the checksum function is obfuscated using simple obfuscation techniques [66]. Particular efforts are devoted to obfuscate the checksum loop because, by analyzing the loop, the attacker could identify the position of the various gadgets. The strategy we adopt is to introduce specific gadgets for obfuscating the logic of checksum computation. More precisely, these gadgets replace some of the existing gadgets and interrupt handlers with new ones. Furthermore, we obfuscate gadgets singularly by introducing dead code, overlapping instructions, and non-trivial pointers computations.

The gadgets we used for normal checksum computation give, as a side effect, an extra advantage for the verifier over the attacker. The presence of aggressive self-modifying code prevents the attacker from using efficient code emulations techniques, such as dynamic binary translation and software-based virtualization. Indeed, self-modifying code invalidates cached translated code, and forces the emulator to analyze and translate the code again and again. We have experienced directly this problem during the development of Conqueror: self-modifying code executed in system mode caused our development system, based on VirtualBox [125], to trash.

4.4 Evaluation

4.4.1 Prototype

We implemented a prototype of Conqueror to evaluate the effectiveness of our proposed solution. The prototype is specific for untrusted 32-bit systems running Microsoft Windows XP, and it consists in a hybrid user/kernel space component, implementing the verifier protocol, and a device driver that stays resident on the untrusted system.

When the verifier wants to bootstrap a tamper-proof execution environment on the untrusted system, it generates a new checksum function and encrypts it. Checksum functions are generated by leveraging a code generation module, currently written in Python. The verifier uses a kernel component to precisely

measure packets transmission and arrival times. As it will be clear in the following, network delay is estimated with the help of a trusted system located in the same network of the untrusted machine. The kernel component running on the untrusted system passively waits for challenges. When challenged, it fills the TPEB with the encrypted checksum function; when the key is received, the attestation begins. To minimize network latency, both parties intercept challenge requests and responses through a hook installed in the network driver.

To experiment the feasibility of attacks based on hardware-assisted virtualization and their cost we also implemented a minimalistic hypervisor, inspired by the Blue Pill hypervisor [103], that simply resumes normal execution after traps. Obviously, any meaningful hypervisor must be much more sophisticated than this.

4.4.2 Experimental setup

For our experiments we employed three laptops with the following characteristics: Intel Core2 Duo 2.1GHz, with 4GB RAM, and a Broadcom BCM5906M network card, connected on the same 100Mbps local network. The first laptop was used as a verifier, the second one as the untrusted system, and the third one as a trusted system. Since our current implementation does not support SMP, on the laptops we used as trusted and untrusted systems we disabled the secondary core of the CPU. In our experiments, the total size of the TPEB and the executable was fixed to six 4Kb pages.

4.4.3 Estimating the parameters of the challenge

To estimate the various parameters involved in the attestation scheme, we considered two attack scenarios: a dynamic hypervisor-based attack, and a static attack aiming to reverse engineer the checksum function.

To understand how the various parameters of the challenge influenced the overall time to compute the checksum and to understand the opportunities of the attackers, we generated multiple checksum functions, varying the number and type of gadgets and the number of iterations of the checksum loop. After several experiments we decided to fix a minimum for the number of gadgets for "hypervisor detection". In each of the checksum functions we subsequently generated, at least 5% of the total of gadgets performed hypervisor detection.

In order to estimate the maximum checksum computation time and the network round-trip time (RTT), the verifier relies on a third-party trusted system, with the same hardware characteristics of the untrusted system. It is worth noting that checksum functions can be generated ahead of time and their execution time can be precomputed. Indeed, the running time depends only on the checksum function, on the CPU, and on the amount of data to attest. Given multiple measurements of the checksum computation time, we estimate the maximum computation time using Chebyshev's inequality, that states that for a random variable

X, with mean value μ and standard deviation σ, $Pr(\mu-\sigma \leq X \leq \mu+\sigma) \geq 1-\frac{1}{\lambda^2}$, where $\lambda \in \mathbb{R}$. In our context, X is the computation time, including the network RTT[1]. Therefore, the upper bound on checksum computation time is $\Delta_t = \mu+\lambda\sigma$, with confidence $\frac{1}{\lambda^2}$. Similarly, the minimum checksum computation time of the most powerful attacker (i.e., an attacker operating in hypervisor mode) is $\mu-\lambda\sigma$; in the calculation of the minimum computation time of the attacker we assumed the adversary to have a null network overhead.

The number of iterations of the checksum loop must be selected to force the time overhead suffered by the attacker to skyrocket. On the other hand, an excessive number of iterations would increase attacker's opportunities to reverse engineer the checksum function. The challenge is to find the best balance between the two. The approach we used was to generate multiple checksum functions, and to compare the time to compute the checksum in the trusted environment and in the environment controlled by the most powerful attacker. Figure 4.6 depicts the time overhead suffered by the attacker during our simulations, performed using five different checksum functions. More precisely, the figure shows the difference between the time to compute the checksum on the simulated untrusted system and on the trusted one. The simulation confirmed our hypothesis: the time overhead suffered by the attacker increases with the number of iterations of the checksum loop. According to our simulation two iterations are sufficient to detect an attack in our experimental scenario (attestation of six memory pages). However, to prevent false negatives, we doubled the number of iterations. Note that the number of iterations to detect a forgery is inversely proportional to the amount of memory to attest; thus, the number of iterations performed by the checksum loop can be tuned accordingly.

4.4.4 Experimental results

Using the approaches described in the previous paragraphs we generated multiple challenges and used them to verify the effectiveness of Conqueror at detecting authentic checksum computations from forgeries. For clarity we refer to Δ_t, the upper bound of the checksum computation time estimated using Chebyshev's inequality, as the *attacker detection threshold*. In our experiments we chose $\lambda = 11$ to obtain an attacker detection rate with 99% confidence. For each challenge we estimated the attacker detection rate by challenging multiple times the trusted host. Subsequently we challenged the untrusted system twice: once the untrusted host simulated a genuine system (i.e., with no attacker), and once the host simulated the presence of the most powerful dynamic attacker (i.e., an attacker attempting to forge the checksum using a hypervisor-based attack). In all the challenges the untrusted system computed the correct checksum without exceeding the attacker detection rate. Similarly, in all the challenges the untrusted

[1]Clearly attestation requires RTT to be minimal. The verifier can measure the RTT and wait to start the challenge if the RTT is too high.

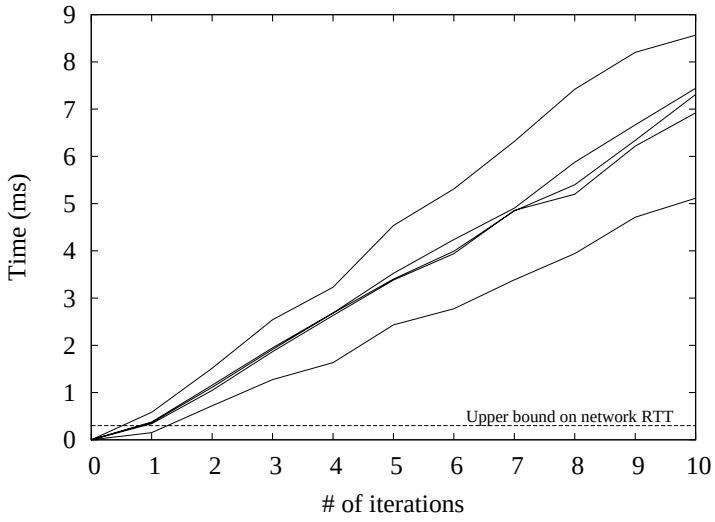

Figure 4.6: Time overhead in a hypervisor-based attack

system under the control of the attacker did not compute the correct checksum in time to be considered authentic.

Figure 4.7 shows the details of one of the challenges we used during the experiment. The figure compares the time the untrusted system took to compute the checksum in the two aforementioned scenarios (the same challenge was repeated more than 50 times). Moreover, the figure shows the attacker detection threshold (Δ_t), and the lower bound for the most powerful attacker ($\mu_{\text{hvm}} - 11\sigma_{\text{hvm}}$). For the challenges in the figure, the average network RTT was less than 0.32ms, and the attacker detection rate was 112.44ms. Similarly, the lower bound for the computation of forged checksum was 115.56ms. The four ms difference and the very small variance between the two clearly indicate that false negatives are practically impossible. The data in the figure confirms the claim: no checksum was forged in time to be considered valid and no authentic checksum was considered forged.

The figure also compares the time requested to compute genuine checksums with the time the attacker would require to perform a preliminary static analysis (i.e., a recursive disassembly) of the checksum function. To measure the cost of the analysis, we loaded in Ida Pro [49], a widely used and well recognized disassembler, the checksum function and then measured the analysis time. Note that the checksum analyzed through Ida Pro was generated without employing any obfuscation technique because the disassembler would not have been able to analyze the code otherwise. The preliminary analysis took about 105ms, just four ms less than the attacker detection rate. Considering that disassembly is fundamental for any static analysis, and that any meaningful analysis to reconstruct the semantic of the checksum function costs much more, it is practically impossible for an attacker to forge a checksum without being detected.

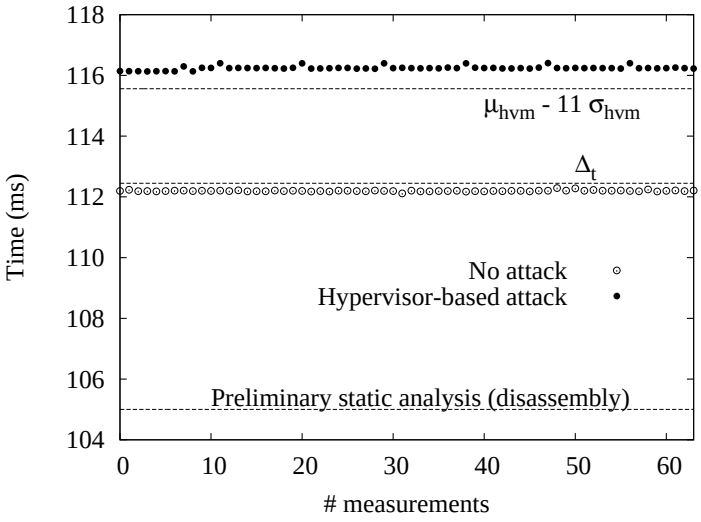

Figure 4.7: Checksums computation time in different scenarios

4.4.5 A real application of Conqueror

Conqueror has been developed to build security applications that must be installed and executed on an untrusted system. All the aforementioned experiments were performed using dummy executables. Nevertheless, Conqueror is an integral part of HyperSleuth, the live forensic analysis tool we describe in Chapter 5. In this case, the goal is to use this loader to install a *measured hypervisor* on an untrusted system [47], on-the-fly, and to segregate the untrusted system in a guest virtual machine. Using Conqueror, we successfully installed HyperSleuth's hypervisor on our test untrusted environment and then resumed the normal, but controlled, execution of the system. In conclusion, Conqueror represents a pure software alternative to the `senter` and `skinit` operations available in the Intel LaGrande [47] and AMD Pacifica [2] technologies for hypervisors secure *late launch*.

4.5 Discussion

Conqueror conservatively assumes that if a hypervisor is installed on the system, then the hypervisor is malicious. It would be worthless to use Conqueror in a system that already runs as a guest of a benign hypervisor: the dynamic root of trust could be established directly by the hypervisor.

The major limitation of Conqueror is the impossibility to bootstrap a tamper-proof environment on SMP and SMT systems. Most modern systems support symmetric flows of executions. An attacker could use the secondary computational resources to forge checksums or to regain control of the execution after attestation. Although we have not addressed the problem in detail, we would like to sketch a possible solution. The verifier can challenge the untrusted SMP (or

SMT) system with multiple challenges simultaneously. More precisely, each processor is given a different checksum function to execute. To solve the challenge, the untrusted system has to compute all the checksums and send them back to the verifier, within the given time frame. Thus, the attacker is left with no spare computational resource to use.

5

Live and trustworthy forensic analysis

The goal of computer forensics is to explain the current state of a computer system or a digital media [136]. Forensic investigations are typically connected with the collection of evidences that might be used in a court. However, in a more general sense, computer forensics aims to acquire some kind of information from a machine, with the guarantee that the acquisition process itself does not alter the data that is being collected. In our context, we are interested in collecting volatile data (e.g., the content of the RAM, the list of active processes) from an allegedly compromised host, and to ensure that even malware that controls the kernel of the operating system cannot tamper with the acquisition process. The data we collect can later be used to detect the presence of malicious programs on the machine. Besides being extremely useful for malware detection at the end-host, a similar solution could also be employed in other situations, such as for the live acquisition of digital evidences for legal proceedings.

In this chapter we address the problem of the live acquisition of volatile data from alleged compromised production systems. In particular, we describe HyperSleuth, a solution that exploits the VMM extensions available nowadays in commodity hardware, to *securely perform live forensic analyses*. HyperSleuth is executed on systems that are believed to be compromised, and obtains complete and tamper-resistant control over the OS, by running in root-mode (i.e., the hypervisor privilege level). HyperSleuth consists in (I) a tiny hypervisor, based on the framework described in Chapter 3, that performs the analysis and (II) a secure loader (Chapter 4) that installs the hypervisor and verifies that its code is not tampered during installation. Like in virtualization-based malware, the *hypervisor is installed on-the-fly*: the alleged compromised host OS is transformed into a guest as it runs [103]. Since the hardware guarantees that the hypervisor is not accessible from the guest code, HyperSleuth remains persistent in the system for all the time necessary to perform the live analysis. On the contrary, other solutions proposed in literature for executing verified code in untrusted environments are not persistent and thus cannot guarantee that the verified code is not

tampered when the execution of the untrusted code is resumed [78, 111, 112]. By providing a persistent trusted execution environment, HyperSleuth opens new opportunities for live and trusted forensic analyses, including the possibility to perform analyses that require to monitor the run-time behavior of the system. When the live analysis is concluded positively (e.g., no malicious program is found), HyperSleuth can be removed from the system and the OS, which was temporarily transformed into a guest OS, becomes again the host OS. As for the installation, the hypervisor is removed on-the-fly.

We developed a memory acquisition tool, a lie detector [42], and a system call tracer on top of HyperSleuth, to show how our hardware-supported VMM-based framework can be successfully used to gather volatile data even from production systems whose services cannot be interrupted. To experimentally demonstrate our claims about the effectiveness of HyperSleuth, we simulated two scenarios: a compromised production system running a heavy-loaded DNS server and a system infected by several kernel-level malware. We used HyperSleuth to dump the content of the physical memory of the former and to detect the malware in the latter. In the first case, HyperSleuth was able to dump the entire content of the physical memory, without interrupting the services offered by the server. In the second case, HyperSleuth detected all the infections.

5.1 Overview

HyperSleuth should not be considered merely as a forensic tool, but rather as a framework for constructing forensic tools. Indeed, its goal is to provide a trusted execution environment to securely perform any live forensic analysis on production systems. More precisely, the execution environment in which a forensic analysis should be performed must guarantee four fundamental properties. First, the environment must guarantee a *tamper-proof* execution of the analysis code. That is, an attacker controlling the system cannot interfere with the analysis and cannot tamper with the results. Second, it must be possible to perform an *a posteriori bootstrap* of the trusted execution environment, even after the system has been compromised, and the bootstrap process itself must require no specific support from the system. Third, the trusted execution environment must be completely *transparent* to the system and to the attacker. Fourth, the trusted execution environment must be *persistent*. That is, the analysis performed in the trusted environment can be periodically interrupted, and the normal execution of the system resumed. Practically speaking, that allows to analyze an alleged compromised system without freezing it and without interrupting the services it provides. Moreover, such a property would allow to perform forensic analyses that require to monitor the run-time behavior of the system. As we will briefly see in the next sections, HyperSleuth fulfills all the aforementioned properties and can thus be used to safely analyze any compromised system that meets the requirements described in Section 5.1.3.

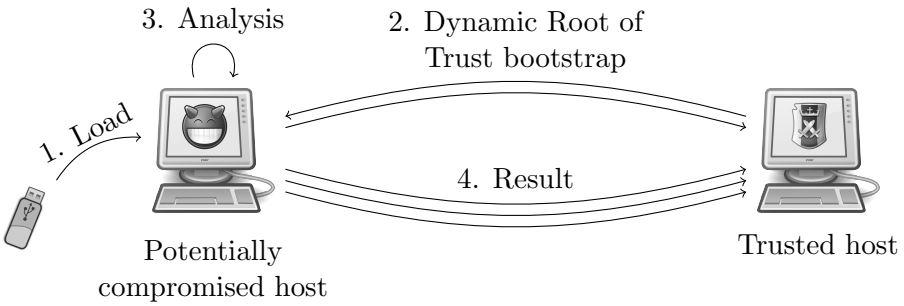

Figure 5.1: Overview of HyperSleuth execution

Figure 5.1 depicts the execution of HyperSleuth. HyperSleuth is installed and executed on demand (step 1 in Figure 5.1), only when there is a suspicious that the host has been compromised, or in general when there is the necessity to perform a live forensic analysis. The execution is characterized by two phases. In the first phase (step 2 in Figure 5.1), HyperSleuth assumes complete control of the host and establishes a Dynamic Root of Trust (DRT). That is accomplished with the collaboration of a trusted host (located in the same local network). The trusted host is responsible for attesting that the DRT has been correctly established. In the second phase (steps 3–4 in Figure 5.1), HyperSleuth performs a specific live forensic analysis and transmits the results of the analysis to the trusted host. Since the trusted host has a proof that the DRT has been correctly established and since, in turn, the DRT guarantees that the analysis code executes in the untrusted host untampered, the results of the analysis can be transitively considered authentic.

In the following, we briefly describe the architecture of HyperSleuth and how it manages to assume and maintain complete control of the untrusted host. Then, we describe the mechanism we use to bootstrap the dynamic root of trust, and, finally, we describe the assumptions and the threat model under which HyperSleuth runs.

5.1.1 HyperSleuth architecture

HyperSleuth needs to be isolated from the host OS, to prevent any attack potentially originating from a compromised system. Simultaneously, HyperSleuth must be able to access certain resources of the host, to perform the requested forensic analysis, and to access the network to transmit the result to the trusted machine.

Figure 5.2 shows the position where HyperSleuth resides in the host. Since HyperSleuth needs to obtain and maintain complete control of the host and needs to operate with more privileges than the attacker, it resides at the lowest level: between the hardware and the host OS. In other words, it executes at the privilege level of a Virtual Machine Monitor (VMM) and thus it has direct access to the hardware and its isolation from the host OS is facilitated by the CPU. At this aim, HyperSleuth is designed as an extension of the VMM-based framework we

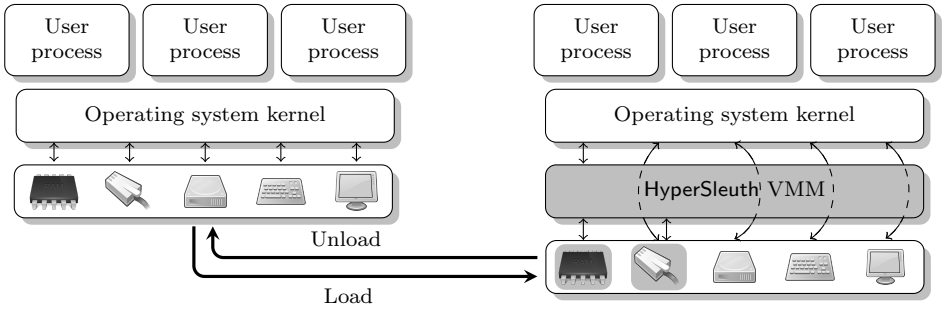

Figure 5.2: Overview of HyperSleuth architecture

propose in Chapter 3. By leveraging hardware virtualization, HyperSleuth can transparently take over an allegedly compromised system, turning, *on-the-fly*, its host OS into a guest one, and vice-versa at will. This is done without rebooting the system and thus preserving all those valuable run-time information that can allow to discover a malware infection or an intrusion.

The greyed portions in Figure 5.2 represent the trusted components in our system. During the launch, HyperSleuth assumes complete control of virtual memory management, to ensure that the host OS cannot access any of its private memory locations. Moreover, HyperSleuth does not trust any existing software component of the host. Rather, it contains all the necessary primitives to inspect directly the state of the guest and to dialog with the network card to transmit data to the trusted party.

Depending on the type of forensic analysis, the analysis might be performed immediately after the launch, or it might be executed in multiple rounds, interleaved with the execution of the OS and users' applications. The advantage of the latter approach over the former is that the host can continue its normal activity while the analysis is being performed. Thus, the analysis does not result in a denial of service and can also target run-time evolving characteristics of the system. In both cases, when the analysis is completed, HyperSleuth can be disabled and even unloaded.

5.1.2 HyperSleuth trusted launch

HyperSleuth's launch process consists in enabling the VMM privilege level, in configuring the CPU to execute HyperSleuth code at this level, and in configuring the CPU such that all virtual memory management operations can be intercepted and supervised by the VMM. Unfortunately, an attacker could easily tamper with the launch. For example, she could simulate a successful installation of the VMM and then transmit fake analysis results to the trusted host. This weakness stems from the fact that the launch process just described lacks an initial trusted component on which we can rely to establish the DRT.

The approach we use to establish the DRT is based on Conqueror, the primitive for tamper-proof code execution described in Chapter 4. Briefly, this primitive

allows to create and to prove the establishment of a minimalistic trusted execution environment that guarantees that the code executed in this environment runs with maximum available privileges and that no attacker can manipulate the code before and during the execution. We use this primitive to create the environment to launch HyperSleuth and to prove to the trusted host that we have established the missing trusted component and that all subsequent operations are secured.

Alternatively to our pure software solution, a TPM-based hardware attestation primitive can be used for this purpose (e.g., Intel `senter` and AMD `skinit` primitives [2, 47]).

5.1.3 Requirements and threat model

Since HyperSleuth leverages hardware support for virtualization available in commodity CPUs, such support must be available on the system that must be analyzed[1]. To maximize the portability of HyperSleuth, we have designed it to only require first generation of hardware facilities for virtualization (i.e., HyperSleuth does not require extensions for MMU and I/O virtualization). Clearly, HyperSleuth cannot be used on systems on which virtualization support is already in use [12]. If a trusted VMM were already running on the host, the VMM could be used directly to perform the analysis. On the other side, if a malicious VMM were running on the host, HyperSleuth's trusted launch would fail.

In order to launch HyperSleuth some privileged instructions must be executed. That can be accomplished by installing a kernel driver in the target host. Note that, in the unlikely case of a damaged system that does not allow to load any kernel driver, alternative solutions for executing code in the kernel can be used (e.g., the page-file attack [103]).

The threat model under which HyperSleuth operates takes into consideration a very powerful attacker, e.g., an attacker with kernel-level privileges. Nonetheless, some assumptions were made while designing HyperSleuth. In particular, the attacker does not operate in system management mode, the attacker does not perform hardware-based attacks (e.g., a DMA-based attack), and the attacker does not leverage an external and more powerful host to simulate the bootstrap of the DRT. Some of these assumptions could indeed be relaxed by virtualizing completely I/O devices using either a pure-software approach or recent hardware support for devices virtualization (e.g., Intel VT-d), and by employing an hardware trusted platform for code attestation (e.g., TPM), keeping HyperSleuth a secure and powerful framework for performing forensic analysis of live data.

[1] Although nowadays all consumer CPUs come with hardware support for virtualization, in order to be usable, the support must be enabled via the BIOS. At the moment we do not know how many manufactures enable the support by default.

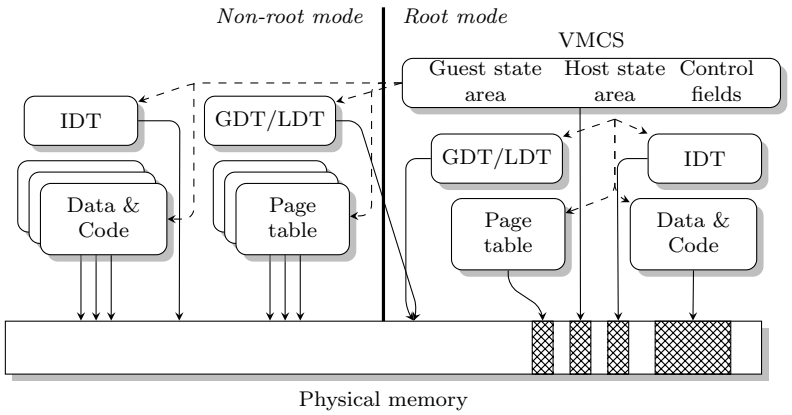

Figure 5.3: Memory layout after the launch of HyperSleuth; ⇢ denotes the CPU contexts stored in the VMCS, ⟶ denotes physical memory mappings, and ▩ denotes the physical memory locations of the VMM that must not be made accessible to the guest.

5.2 Implementation

The core of HyperSleuth is the framework we described in Chapter 3. In this section we will focus on the peculiar characteristics of HyperSleuth that are not part of the framework discussed in the previous chapter.

5.2.1 HyperSleuth VMM

HyperSleuth can be loaded at any time by exploiting the delayed launch feature offered by the CPU. Figure 5.3 shows a simplified memory layout after the launch of HyperSleuth. The environment for non-root mode, in which the OS and users' application are executed, is left intact. The environment for root mode instead is created during the launch and maintained isolated by the VMM. In the following paragraphs we describe the steps required to securely launch the VMM, to enforce the isolation of root mode from non-root mode, and to access hardware peripherals.

VMM trusted launch and removal

To launch HyperSleuth VMM in a running host we perform the following operations. First, we allocate a fixed-size chunk of memory to hold the data and code of the VMM. Second, we enable VMX root-mode. Third, we create and initialize the VMCS. Fourth, we resume the normal execution of the guest by entering non-root mode.

Although on the paper the launch of the VMM appears a very simple process, it requires to perform several operations. Such operations must be performed atomically, otherwise a skilled attacker may interfere with the whole bootstrap

process and tamper with VMM code and data. To maximize HyperSleuth portability, we decided to address this problem using the software-based primitive for tamper-proof code execution we thoroughly describe in Chapter 4. In summary, the primitive is based on a challenge-response protocol and a checksum function. The trusted host issues a challenge for the untrusted system and the challenge consists in computing a checksum. The result of the checksum is sent back to the trusted host. A valid checksum received within a predefined time is the proof that a Trusted Computing Base (TCB) has been established on the untrusted system. The checksum function is constructed such that the correct checksum value can be computed in time only if the checksum function and the code for launching the VMM are not tampered, and if the environment in which the checksum is computed and in which the VMM launch will be performed guarantees that no attacker can interrupt the execution and regain the control of the execution before the launch is completed. Practically speaking, the correct checksum will be computed in time only if the computation and the launch are performed with kernel privileges, with interrupts disabled, and no VMM is running.

At the end of the analysis, HyperSleuth can be completely removed from the system. The removal essentially is the opposite process of the launch. We start by disabling VMX root-mode, then we deallocate the memory regions assigned to the VMM (e.g., the Interrupt Descriptor Table, the stack, and the code). Finally, we update the context of the CPU such that the OS and users' applications can resume their normal execution.

MMU virtualization

In order to guarantee complete isolation of the VMM from the guest, it is essential to ensure that the guest cannot access any of the memory pages in use by the VMM (i.e., the crosshatched regions in Figure 5.3). However, to perform any useful analysis, we need the opposite to be possible.

Although modern x86 CPUs provide hardware support for MMU virtualization, we have opted for a software-based approach to maximize the portability of HyperSleuth. The approach we use is based on the assumption that the direct access to physical memory locations is not allowed by the CPU (with paging enabled) and that physical memory locations are referenced through virtual addresses. The CPU maintains a mapping between virtual and physical memory locations and manages the permissions of these locations through page tables. By assuming the complete control of the page tables, the VMM can decide which physical locations the guest can access. To do that, the VMM maintains a *shadow page table* for each page table used by the guest, and tricks the guest into using the shadow page table instead of the real one [118].

A shadow page table is a clone of the original page table and is used to maintain a different mapping between virtual and host physical addresses and to enforce stricter memory protections. In our particular scenario, where the VMM manages a single guest and the OS has already filled the page tables (because

the VMM launch is delayed), the specific duty of the shadow page table is to maintain as much as possible the original mapping between virtual and physical addresses and to ensure that none of the pages assigned to the VMM is mapped into a virtual page accessible to the guest. As described in Section 5.3, we also rely on the shadow page table to restrict and trap certain memory accesses to perform the live forensic analysis. The algorithm we currently use to maintain the shadow page tables trades off performance for simplicity and is based on tracing and simulating all accesses to tables.

Unrestricted guest access to I/O devices

In the typical deployment, physical I/O devices connected to the host are shared between the VMM and one or more guests. In our particular scenario, instead, there is no need to share any I/O device between the guest and the VMM: HyperSleuth executes batch and interacts only with the trusted host via network. Thus, the guest can be given direct and unrestricted access to I/O devices. Since the OS runs in non-root mode, unmodified, and at the highest privilege level, it is authorized to perform I/O operations, unless the VMM configures the execution control fields of the VMCS such that I/O operations cause exits to root-mode. By not doing so, the VMM allows the guest OS to perform unrestricted and direct I/O. This approach simplifies drastically the architecture of the VMM and, most importantly, allows the OS to continue to perform I/O activities exactly as before, without any additional overhead.

Direct network access

HyperSleuth relies on a trusted host to bootstrap the dynamic root of trust and to store the result of the analysis. Since we are assuming that no existing software component of the host can be trusted, the only viable approach to communicate securely over the network is to dialog directly with the network card. For this reason, HyperSleuth contains a minimalistic network driver that supports the card available on the host. All the data transmitted over the network is encapsulated in UDP packets. Packets are signed and encrypted automatically by the driver using a pre-shared key, which we hardcode in HyperSleuth just before the launch.

As described in the previous paragraph, HyperSleuth does not virtualize hardware peripherals, but it lets the guest to access them directly. Thus, the network card must be shared transparently with the guest. In other words, to avoid interferences with the network activity of the guest, HyperSleuth must save and restore the original state of the card (i.e., the content of PCI registers), respectively before and after using the network. To transmit a packet the driver writes the physical address and the size of the packet to the appropriate control registers of the device. The driver then polls the status register of the device until the transmission is completed. Polling is used because, for simplicity, we execute all VMM code with interrupts disabled. Packets reception is implemented in the same way.

5.3 Live forensic analysis

HyperSleuth operates completely in batch mode. The only user action required is to copy an executable on the system to be analyzed and to fire its execution. This executable is a loader that establishes the dynamic root of trust by creating a tamper-proof execution environment and by using this environment to launch the VMM. Note that the loader is removed from the memory and the disk to prevent malicious software to detect its presence. Once launched, the VMM performs the forensic analysis, transmits the results to the trusted hosts and then removes itself.

Although HyperSleuth VMM is completely transparent to the OS and users' applications and it is removed after the end of the analysis, the launch of the VMM is a slightly invasive process. Indeed, it requires to execute the loader that in turn loads a kernel driver (to launch the VMM) and might start other additional in-guest utilities. Our claim is that, considered the valuable volatile information HyperSleuth can gather from the system, the little modifications its installation produces to the state of the system are an acceptable compromise. After all, no zero invasive solution for *a posteriori* forensic analysis exists.

Currently, HyperSleuth supports three live forensic applications: a lazy physical memory dumper, a lie detector, and a system call tracer. Clearly, all these analyses could be performed also without the need of a dynamic root of trust and the VMM. Indeed, there are several commercial and open source applications with the same capabilities available, but, by operating at the same privilege level of the OS kernel to analyze, they can easily be tampered by an attacker (with the same privileges), and cannot thus provide the safety guarantees offered by HyperSleuth.

5.3.1 Physical memory dumper

Traditional approaches for dumping the content of the physical memory are typically based on kernel drivers or on FireWire devices. Unfortunately, both these approaches have a major drawback that limits their applicability to non production systems. Dumping the content of the physical memory is an operation that should be performed atomically, to guarantee the integrity of the dumped data. Failing to achieve this would, in fact, enable an attacker to make arbitrary modification to the content of the memory, potentially hampering any forensic analysis of live data. On the other side, if the dump is performed atomically, the system, and the services the system provides, will be blocked for the entire duration of the dump. That is not desirable, especially if there is only a marginal evidence that the system has been compromised. As the dump may be very time consuming, the downtime might be economically very expensive and even dangerous.

To address this problem, we exploit HyperSleuth's persistent trusted execution environment to implement a new approach for dumping lazily the content of the physical memory. This approach guarantees that the state of the physical

```
1  switch (VMM exit reason)
2    case CR3 write:
3       Sync PT and SPT
4       for (v = 0; v < sizeof(SPT); v++)
5          if (SPT[v].Writable && !DUMPED[SPT[v].PhysicalAddress])
6             SPT[v].Writable = 0;
7
8    case Page fault: // 'v' is the faulty address
9       if (PT/SPT access)
10         Sync PT and SPT and protect SPTEs if necessary
11      else if (write access && PT[v].Writable)
12         if (!DUMPED[PT[v].PhysicalAddress])
13            DUMP(PT[v].PhysicalAddress);
14         SPT[v].Writable = DUMPED[PT[v].PhysicalAddress] = 1;
15      else
16         Pass the exception to the OS
17
18   case Hlt:
19      for (p = 0; p < sizeof(DUMPED); p++)
20         if (!DUMPED[p])
21            DUMP(p); DUMPED[p] = 1;
22            break;
```

Figure 5.4: Algorithm for lazy dump of the physical memory

memory dumped corresponds to the state of the memory *at the time the dump is requested*. That is, no malicious process can "clean" the memory after HyperSleuth has been installed. Moreover, being performed lazily, the dump of the state of the memory does not monopolize the CPU and does not interrupt the execution of the processes running in the system. In other words, HyperSleuth allows to dump the content of the physical memory even of a production system without causing any outage of the services offered by the system.

The dump of the memory is transmitted via network to the trusted host. Each page is fragmented, to fit the MTU of the channel, and labelled. The receiver reassembles the fragments and reorders the pages to reconstruct the original bitstream image of the physical memory. To ease further analysis, the image produced by HyperSleuth is compatible with off-the-shelf tools for memory forensic analysis (e.g., Volatility [135]).

The algorithm we developed for dumping lazily the content of the physical memory is partially inspired by the technique used by operating systems for handling shared memory and known as *copy-on-write*. The rationale of the algorithm is that the dump of a physical memory page can be safely postponed until the page is accessed for writing. More precisely, the algorithm adopts a combination of two strategies to dump the memory: *dump-on-write* (DOW), and *dump-on-idle* (DOI). The former permits to dump a page before it is modified by the guest; the latter permits to dump a page when the guest is idle. Note that the algorithm assumes that the guest cannot access directly the physical memory. However, an attacker could still program a hardware device to alter the content of the memory by performing a DMA operation. In our current threat model we do not consider DMA-based attacks.

Figure 5.4 shows the pseudo-code of our memory dumper. Essentially the VMM intercepts three types of events: updates of the page table address, page-

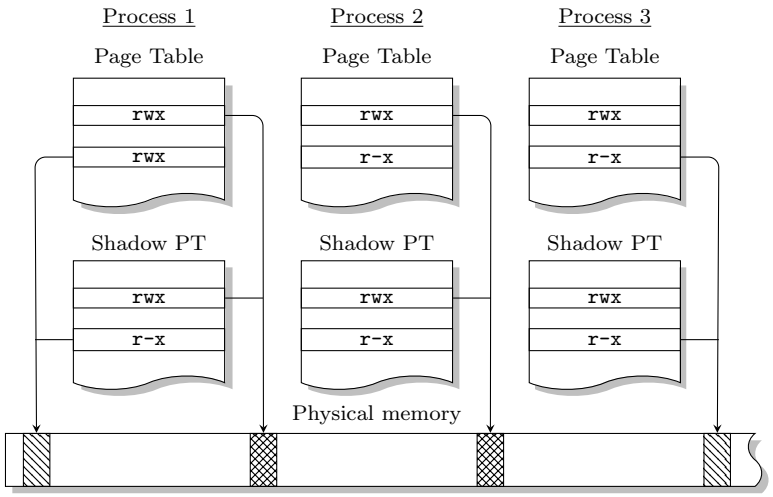

Figure 5.5: Overview of permissions used to implement dump-on-write (▩ and ▧ denote respectively dumped and not dumped physical pages)

fault exceptions, and CPU idle loops. The algorithm maintains a map of the physical pages that have already been dumped (`DUMPED`) and leverages the shadow page table (`SPT`) to enforce stricter permissions than the ones specified in the real page table (`PT`) currently used by the system. When the page table address (stored in the `CR3` register) is updated, typically during a context switch, the algorithm synchronizes the shadow page table and the page table (line 3). Subsequently, all the entries of the shadow page table mapping physical not yet dumped pages are granted read-only permissions (lines 4–6). Such a protection ensures that all the memory accesses performed by the guest OS for writing to any virtual page mapped into a physical page that has not been dumped yet result in a page fault exception. The VMM intercepts all the page fault exceptions for keeping the shadow page table and the real page table in sync, for reinforcing our write protection after every update of the page table (lines 9–10), and also for intercepting all write accesses to pages not yet dumped (lines 11–14). The latter type of faults are characterized by a write access to a non-writable virtual page that is marked as writable in the real page table. If the accessed physical page has not been dumped yet, the algorithm dumps the page and flags it as such. All other types of page fault exceptions are delivered to the guest OS that will manage them accordingly. Finally, the VMM detects CPU idle loops by intercepting all occurrences of the `hlt` instruction. This instruction is executed by the OS when there is no immediate work to be done, and it halts the CPU until an interrupt is delivered. We exploit these short idle periods to dump the pending pages (lines 19–22). It is worth noting that a loaded system might enter very few idle loops. For this reason, at every context switch we check whether the CPU has recently entered the idle loop and, if not, we force a dump of a small subset of the pending pages (not shown in the figure).

Figure 5.5 shows the protections enforced by the algorithm, through the

shadow page table, to trap all write accesses to the physical memory pages that have not been dumped yet.

5.3.2 Lie detector

Kernel-level malware is particularly insidious as it operates at a very high privilege level and can, in principle, hide any resource an attacker wants to protect from being discovered (e.g., processes, network communications, files). Different techniques exist to achieve such a goal (see [51, 6]), but all of them aim at forcing the OS to lie about its state, eventually. Therefore, the only effective way to discover such liars is to compare the state of the system perceived from the system itself with the state of the system perceived by a VMM. Unfortunately, so far lie detection has been possible only using a traditional VMM and thus it has not been applicable on production systems not already deployed in virtual machine environments. On the other hand, HyperSleuth's hot-plug capability of securely migrating a host OS into a guest one (and vice-versa) on-the-fly makes it a perfect candidate for detecting liars in production systems that had not been deployed in virtual machine environments since the beginning.

To this end, besides launching the VMM, HyperSleuth loader runs a simple in-guest utility that collects detailed information about the state of the system and transmits its output to the trusted host. This utility performs the operations typically performed by system tools to display information about the state of the system and intentionally relies on the untrusted code of the OS. The intent is to trigger the malicious code installed by the attacker to hide any malicious software component or activity. For example, this utility collects the list of running processes, active networks connections, loaded drivers, open files and registry keys, and so on. At the end of its execution, the utility performs a VMM call to transfer the execution to the HyperSleuth VMM. At this point the VMM collects the same information through OS-aware inspection. That is, the VMM does not rely on any untrusted code of the system, but rather implements its own primitives for inspecting the state of the guest and, when possible, offers multiple primitives to inspect the state of the same resource. For example it offers primitives to retrieve the list of running processes/threads, each of which relies on a different data structure available in the kernel. Finally, the trusted host compares the views provided by the in-guest utility and the VMM.

Since the state of the system changes dynamically and since the in-guest utility and the VMM does not run simultaneously, we repeat the procedure multiple times, with a variable delay between each run to limit any measurement error.

5.3.3 System call tracer

System calls tracing has been widely recognized as a way to infer, observe, and understand the behavior of processes [35]. Traditionally, system calls were invoked by executing software interrupt instructions causing a transition from user-space

to kernel-space. Such user-/kernel-space interactions can be intercepted by HyperSleuth, as interrupt instructions executed by the guest OS in VMX non-root mode cause an exit to VMX root mode, i.e., to the VMM.

Alternative and more efficient mechanisms for user-/kernel-space interactions have been introduced by CPU developers, recently. Unfortunately, Intel VT-x does not support natively the tracing of system calls invoked through the fast invocation interface used by modern operating systems (`sysenter`/`sysexit`). The approach we use to trace system calls is thus inspired by Ether [29]. System calls are intercepted through another type of exits: synthetic page fault exceptions. All system calls invocations go through a common gate, whose address is defined in the `SYSENTER_EIP` register. We shadow the value of this register and set the value of the shadow copy to the address of a non-existent memory location, such that all system calls invocations result in a page fault exception and in an exit to root mode. The VMM can easily detect the reason of the fault by inspecting the faulty address. When a system call invocation is trapped by the VMM, it logs the system call and then resumes the execution of the guest from the real address of `SYSENTER_EIP`. To intercept returns from system calls we mark the page containing the return address as not accessible in the shadow page table.

When a system call is invoked or returns, HyperSleuth retrieves and parses all the input and output arguments of the call and includes them in the trace. Moreover, HyperSleuth includes in the trace information about the process and the thread involved in the call. Like with all the results of other types of analysis, the trace is transmitted via network to the trusted host.

5.4 Experimental evaluation

We implemented a prototype of HyperSleuth and of the routines for the three analyses described in Section 5.3. Our implementation is an extension of the framework discussed in Chapter 3. For the secure loader we currently use Conqueror, the attestation primitive presented in Chapter 4. While the core of HyperSleuth is mostly OS-independent, the routines for the analysis (e.g., the enumeration of running processes and of active network connections) are OS-dependent and may require to be slightly adapted to provide support for different operating systems.

In this section we discuss the experimental results concerning the launch of HyperSleuth, the lazy physical memory dumper, and the lie detector. To this end, we simulated the compromised production system using an Intel Core i7, with 3GB RAM, and a Realtek RTL8139 100Mbps network card. Note that we disabled all cores of the CPU but one, since the VMM currently supports a single core. We simulated the trusted host using a laptop. We used the trusted host to attest the correct establishment of the dynamic root of trust and to collect and subsequently analyze the results of the analysis.

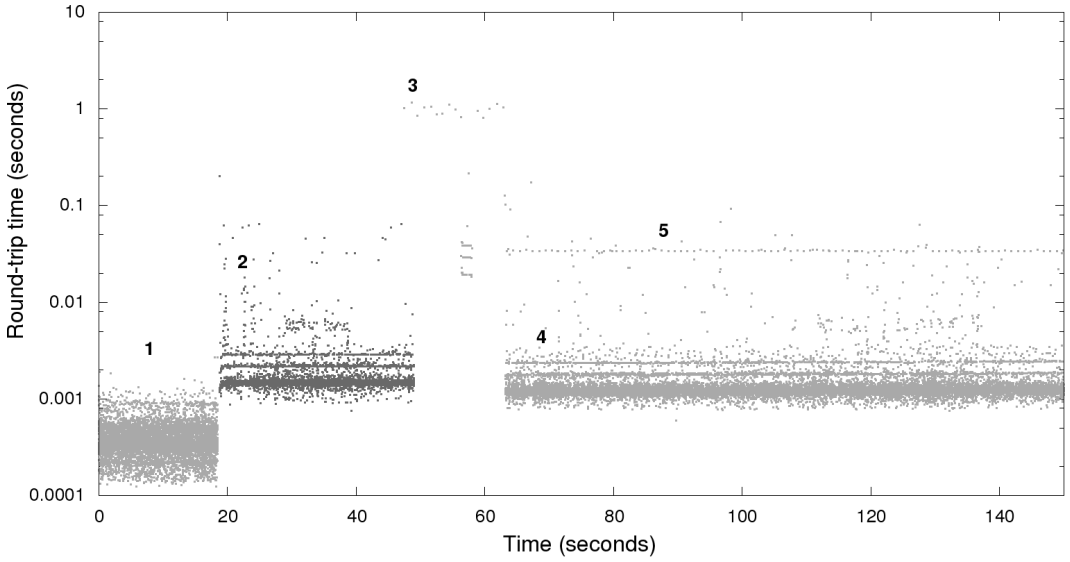

Figure 5.6: Round-trip time of the queries performed against the compromised production DNS server before (1) and after (2) the launch of HyperSleuth and (3–5) during the lazy dump of the physical memory (the scale of the ordinate is logarithmic).

5.4.1 HyperSleuth launch and lazy dump of the physical memory

To evaluate the cost of launching HyperSleuth, the base overhead of the VMM, and the cost of the lazy physical memory dumper we simulated the following scenario. A production DNS server was compromised and we used HyperSleuth to dump the entire content of the physical memory when the server was under the heaviest possible load. We used an additional laptop, located on the same network, to flood the DNS server with queries and to measure the instantaneous round-trip time of the queries. About 20 seconds after we started the flood, we launched HyperSleuth; 25 seconds later we started to dump the content of the memory.

Figure 5.6 summarizes the results of our experiments. The graph shows the round-trip time of the queries sent to the compromised DNS server over time. For the duration of the experiment, the compromised machine was able to handle all the incoming DNS queries, and no query timed out. Before launching HyperSleuth the average round-trip time was ~ 0.34ms (mark 1 in Figure 5.6). Just after the launch, we observed an initial increase of the round-trip time to about 0.19s (mark 2 in Figure 5.6). This increase was caused by the bootstrap of the dynamic root of trust and then by the launch of the VMM, which must be performed atomically. After the launch, the round-trip time quickly stabilized around 1.6ms, less than five times the round-trip time without the VMM. The overhead introduced by the VMM was mostly caused by the handling of the shadow page table. When we started the dump of the physical memory we observed another and steeper

peak (mark 3 in Figure 5.6). We were expecting this behavior since there are a lot of writable memory pages that are frequently accessed (e.g., the stack of the kernel and of the user-space processes and the global variables of the kernel) and that, most likely, are written each time the corresponding process is scheduled. Thus, the peak was caused by the massive number of write accesses to pages not yet dumped. A dozen of seconds later the round-trip time stabilized again around 1.6ms (mark 4 in Figure 5.6). That corresponds to the round-trip time observed before we started the dump. Indeed, the most frequently written pages were written immediately after the dump was started, and the cost of the dump of a single page was much less than the round-trip time and was thus unnoticeable. The regular peaks around 32ms about every second (mark 5 in Figure 5.6) were instead caused by the periodic dump of non-written pages. Since the system was under heavy load, it never entered an idle loop. Thus, the dump was forced after every second of uninterrupted CPU activity. More precisely, the dumper was configured to dump 64 physical pages about every second. Clearly, the number of non-written pages to be dumped when either the system enters the idle loop, or the duration of uninterrupted CPU activity hits a certain threshold, is a parameter that can be tuned accordingly to the urgency of the analysis, to how critical the system is, and to the throughput of the network.

In conclusion, the dump of the whole physical memory of the system (3GB of RAM), in the setting just described, required about 180 minutes and the resulting dump could be analyzed using an off-the-shelf tool, such as Volatility [135]. The total time could be further decreased by increasing the number of physical pages dumped periodically, at the cost of a higher average round-trip time. It should also be pointed out that, on a 1Gbps network, we could increase the number of physical pages dumped every second to 640, without incurring in any additional performance penalty. In this case, the whole physical memory (3GB) would be dumped in just ~ 18 minutes. It is important to remark that although HyperSleuth, and in particular the algorithm for dumping lazily the memory, introduces a non-negligible overhead, we were able to dump the entire content of the memory without interrupting the service (i.e., no DNS query timed out). On the other hand, if the memory were dumped with traditional (atomic) approaches the dump would require, in the ideal case, about 24 seconds, 50 seconds, and 4 minutes respectively on a 1Gbps network, on a 480Mbps FireWire channel, and on a 100Mbps network (these estimations are computed by dividing the maximum throughout of the media by the amount of data to transmit). In these cases, the production system would have not been able to handle any incoming request, for the entire duration of the dump.

5.4.2 Lie detection

Table 5.1 summarizes the results of the experiments we performed to assess the efficacy of the lie detection module. To this end, we used nine malware samples, each of which included a root-kit component to hide the malicious activity per-

Sample	Characteristics	Detected?
FU	DKOM	✓
FUTo	DKOM	✓
HaxDoor	DKOM, SSDT hooking, API hooking	✓
HE4Hook	SSDT hooking	✓
NtIllusion	DLL injection	✓
NucleRoot	API hooking	✓
Sinowal	MBR infection, Run-time patching	✓
Smiscer	DKOM, Run-time patching	✓
TDL3	DKOM, Run-time patching	✓

Table 5.1: Results of the evaluation of HyperSleuth's lie detector with nine different malware (all equipped with a root-kit component)

formed on the infected system. We used HyperSleuth's lie detector to detect the hidden activities. The results testify that our approach can be used to detect both user- and kernel-level root-kits.

For each malware sample we proceeded as follows. First, we let the malware infect the untrusted system. Then, we launched HyperSleuth on the compromised host and triggered the execution of the lie detector. The module performed the analysis, first by leveraging the in-guest utility, and then by collecting the same information directly from the VMM through OS-aware inspection. The results were sent separately to the trusted host. On the trusted host we compared the two views of the state of the system and, in all cases, we detected some discrepancies between the two. These discrepancies were all caused by lies. That is, the state visible to the in-guest utility was altered by the root-kit, while the state visible to HyperSleuth VMM was not.

As an example, consider the FUTo root-kit. This sample leverages direct kernel object manipulation (DKOM) techniques to hide certain kernel objects created by the malware (e.g., processes) [51]. Our current implementation of the lie detector counteracts DKOM through a series of analyses similar to those implemented in RAIDE [9]. Briefly, those analyses consist in scanning some internal structures of the Windows kernel that the malware must leave intact in order to preserve its functionalities. Thus, when we compared the trusted with the untrusted view of the state of the system we noticed a process that was not present in the untrusted view produced by the in-guest utility. Another interesting example is NucleRoot, a root-kit that hooks Windows' System Service Descriptor Table (SSDT) to intercept the execution of several system calls and to filter out their results, in order to hide certain files, processes, and registry keys. In this case, by comparing the two views of the state of the system, we observed that some registry keys related to the malware were missing in the untrusted view. Although we have not yet any empirical proof, we speculate the even root-kits like Shadow Walker [122] would be detected by our lie detector since our

approach allows to inspect the memory directly, bypassing a malicious page-fault handler and bogus TLBs' entries.

5.5 Discussion

In this chapter we proposed HyperSleuth as a generic infrastructure to support live and trusted forensic analyses of allegedly compromised systems. To further corroborate our thesis, we built upon its top three different analysis tools. One of the fundamental characteristics of our VMM-based infrastructure is that analysis tools are run in hypervisor-mode, i.e., at a higher privilege level than the monitored operating system. An important consequence is that a tool cannot leverage user-level libraries, nor the programming interface offered by the kernel. Obviously, this limitation complicates the development of analysis tools: HyperSleuth itself includes some primitives to support very common activities (e.g., string operations), but it cannot provide the same functionalities offered by fully-featured programming libraries. A possible solution is to equip the hypervisor with a self-contained implementation of general-purpose libraries, and to implement a minimalist set of hypervisor-level primitives to support their execution.

Finally, so far we presented HyperSleuth from a technical prospective. The decisions we made in designing and implementing HyperSleuth were mostly motivated by the intent of minimizing the dependencies on the hardware and of maximizing the portability. Therefore, we always opted for pure software-based approaches (e.g., to secure the launch of the VMM and to virtualize the MMU), whenever possible. However, since HyperSleuth is a framework for performing live forensic analyses, it is important to reason about its probatory value. From such a prospective, we must take into account that the trustworthiness of the results of the analyses depends on the trust people have in the tool that generated the results. To strengthen its probatory value, all HyperSleuth's components should be verified in order to prove that their code meets all the expectations [37]. At this aim, in the future we plan to further decrease the size of HyperSleuth's code base in order to ease its verifiability (e.g., by leveraging hardware-based attestation solutions, such as the TPM).

6 Related literature

The solutions we discuss in this dissertation are related with several research areas, such as malware analysis, code attestation, and dynamic analysis infrastructures. In this chapter we briefly review the related literature and we highlight existing differences and similarity between previous approaches and our own research work.

6.1 Malware analysis

6.1.1 Behavior-based malware analysis

Behavior-based malware analysis is a very promising approach that recently gained the attention of the research community. In Chapter 1 we already sketched out the problems that affect behavior-based techniques. During the last years, researchers suggested many possible solutions to overcome these limitations. In this section we review some of the recent advances in behavior-based analysis and detection strategies.

Exploration of multiple program paths

One of the main problems of dynamic approaches for malware analysis is their incompleteness. To address this limitations, in [80] Moser *et al.* proposed a system that dynamically monitors a suspicious program to identify the execution points where the application makes control-flow decisions based on input-dependent values. For each of these program points, the system forks the execution to explore both paths. To explore a path different from the one that is intentionally executed by the program, the system tracks the linear dependencies between the variables used in the control flow decisions and the input. Then, a constraint solver is used to generate a configuration of input values that allows to follow the new program path. In [8] Brumley *et al.* describe a multiple path exploration system that is based on a similar approach. The infrastructure we propose in

Chapter 2 addresses the incompleteness problem from a completely different perspective: instead of performing a systematic exploration of all input-dependent program paths, we exploit the high diversity of end-users' environments to induce a malware to exhibit its malicious behaviors. A thorough comparison between our solution and the systems based on multiple path exploration is presented in Section 2.3.

It is worth pointing out that, before being employed for malware analysis, multiple path exploration approaches have been also investigated by the software engineering community to automatically discover bugs, by analyzing either the source code [44, 10] or the binary representation of a program [65, 45].

"Out-of-the-box" malware analysis

In Chapter 5 we investigate the idea of leveraging a virtual machine monitor to perform sophisticated run-time analyses, with the guarantee that the results cannot be tampered by a malicious attacker. This approach has been widely explored in the literature. Garfinkel *et al.* were the first to propose to use a VMM to perform OS-aware introspection [42], and subsequently the idea was further elaborated [96, 54, 29]. Other researchers instead proposed to use a VMM to protect the guest OS from attacks by supervising its execution, both with a software-based VMM [99] and by leveraging hardware support for virtualization [110]. Similar ideas were also suggested by other authors [92, 116]. In [17], Chen *et al.* proposed a solution to protect applications' data even in the presence of a compromised operating system. More recently, Vasudevan *et al.* presented XTREC, a lightweight framework to record securely the execution control flow of all code running in an untrusted system [93]. Unfortunately, in order to guarantee that the analyses they perform cannot be tampered by an attacker, all the existing solutions must take control of the system *before* the guest is booted, and cannot be removed until the guest is shut down. On the contrary, the solution we describe in Chapter 5 can be installed as the compromised system runs, and, when the analyses are completed, it can be removed on-the-fly. The idea to take advantage of the possibility to install a VMM on a running system was also sketched in [105].

Besides using virtual machines to analyze malware and protect the end-users, researchers also proposed to use VMMs to implement malware that are particularly hard to detect and to eradicate. SubVirt was one of the first prototypes that employed this technique [58]. Being implemented using a software-based VMM, the installation of SubVirt required to reboot the machine, and the malware also introduced a noticeable run-time overhead in the infected target. Later, the Blue Pill malware started to exploit the hardware-assisted supports for virtualization to implement an efficient VMM-based malware that is able to infect a machine as it runs, without the need for reboot [103, 27]. Our VMM-based analysis framework was inspired by this malware.

Efficient malware analysis

Behavior-based approaches can be very effective, but effectiveness comes at a price: these approaches introduce a high run-time overhead that prevents them from being used as detection solutions at the end-host. Recently, researchers developed several techniques to improve the efficiency of behavior-based detectors. In [116] Sharif *et al.* introduce a framework that allows "in-VM" monitoring and detection. Their observation is that the approaches that employ virtual machine introspection techniques to isolate security tools from the untrusted environment are very effective, but they are also computationally expensive. For this reason, they propose to place security applications right inside an untrusted system for efficiency, while using hardware-assisted virtualization facilities to protect the "in-VM" detector.

Kolbitsch *et al.* describe a technique for efficient and effective malware detection [62]. Their idea is to build models of the malicious samples off-line, and then to verify at run-time if the behavior of a suspicious application adheres to a known model. The idea of building a model of malicious behaviors has been also investigated by other researchers. As an example, in [21] the authors derive automatically behavioral models by comparing the execution of a malware with a set of benign applications, while in [76] Martignoni *et al.* use hierarchical behavior specifications to build a model of a malicious program. As the number of malicious samples keeps growing, efficiency is essential not only for detectors, but also for automatic malware analysis systems. To address this problem, Bayer *et al.* propose a technique that allows to detect if a binary is a polymorphic variation of a malware sample that has already been analyzed in the past [4] . Their approach consists in comparing the dynamic behavior of a new binary program with a database of known behaviors. If a match is found, the new program needs not to be analyzed.

6.1.2 Malware analysis in the cloud

The approach we discuss in Chapter 2 leverages cloud computing to blend together the computational power available in security laboratories (the cloud) with the heterogeneity of end-users' environments. In the last years, similar ideas were also suggested by other researchers and anti-malware companies in order to provide more comprehensive and effective protection solutions. CloudAV is the first implementation of an in the cloud malware detector through which end-users delegate to a central authority the task of detecting if an unknown program is malicious or not [86]. More recently, a similar approach, called "collective intelligence", has also been introduced in a commercial malware detector [90]. Such centralized detection gives two major benefits. First, the analysis no longer impacts on end-users' systems, and, being centralized, it can be made more fine-grained. For example CloudAV analyzes programs simultaneously with multiple off-the-shelf detectors. Second, the results of the analysis can be cached to serve

future requests of other users at no cost.

We further enhanced the aforementioned solutions by proposing a framework that leverages the systems of potential victims to make the behavioral analysis much more complete. The strategy we adopt to force a program executed in a security lab to behave as in the environment of the end-user involves system calls proxying techniques. In previous research work, remote system call execution has been successfully used to implement a high-throughput computation environment based on Condor [67], where files stored on remote nodes of the environment are made accessible locally and transparently by proxying the appropriate system calls. Similarly, the $\mathcal{V}^2$ project [133] includes support for remote system call execution. The framework we describe in Chapter 2 leverages system call proxying to achieve a completely different goal.

6.1.3 Post-infection countermeasures

Back in 1987 Fred Cohen demonstrated that no algorithm exists to precisely discern malware samples from benign programs [25]. As a consequence, there will always exist some malware that is able to bypass proactive detection solutions and to infect the system. Obviously, the safest way to remediate an infection is to format the permanent storage and re-install the operating system from scratch. While effective, this approach is also costly, time expensive, and usually results in a loss of valuable personal data. Rather than re-installing compromised systems from scratch, alternative solutions have been presented.

Several researchers proposed to use sandboxes to isolate suspicious programs and to prevent damages to end-users' systems. The idea is to execute untrusted programs inside a sandbox, and the changes made to the "virtual" system are committed to the real one at the end of the execution, but only if the program can be considered innocuous. As an example, Sun *et al.* introduced a one-way isolation technique to safely execute untrusted programs [124]. Their approach consists in isolating the effects of an untrusted program from the rest of the system by intercepting system calls that modify the file-system and redirecting them to a cache, invisible to other processes. When the untrusted program terminates, the user can choose to discard these modifications, or to commit them to the real system. The approach we adopt in Chapter 2 to proxy the access to remote system resources is similar to the one proposed by Sun *et al.* A similar technique was also proposed by Hsu *et al.* [52]; the difference is that the execution of the untrusted program is not isolated, but monitored, and at the end of the execution the modifications made to the system can be reverted.

Unfortunately, sandboxing is not very popular, and users typically prefer to leverage remediation capabilities of anti-malware products to revert the effects of an infection. A recent study we performed demonstrated that even top-rated commercial anti-malware software fails to revert the effects of all the actions performed by malware during infections [91]. For this reason, in [57] Kim *et al.* describe RETRO, an infrastructure to repair a system that has been compromised

by a malicious program: RETRO first records a dependency graph that describes the actions taken during normal system execution; then, during repair, the action graph is used to undo the unwanted actions. In [88] we proposed an architecture to automatically generate remediation procedures from malicious programs, i.e., procedures that can be used to remediate all and only the effects of the execution of the malware in *any* infected system. The infrastructure presented in Chapter 5 can be used to take over a compromised system and to support the execution of these remediation procedures, with the guarantee that the malware running on the host cannot affect their execution.

6.2 Code attestation

The majority of the research work on software-based attestation and verifiable code execution is specific for embedded devices and sensor networks. Most of the schemes are based on the same type of challenge and response protocol [112, 109, 108, 111]; we thoroughly presented it in Section 4.1. The strength and weaknesses of these schemes have been studied by Castelluccia *et al.* [22]. Conqueror, the software-based attestation primitive described in Chapter 4, is instead inspired by the work done by Shaneck *et al.* and by Garay *et al.* [39, 113]. However, the two attestation schemes are also specific for embedded devices and not suited at all for attestation on legacy systems. Genuinity and Pioneer are two schemes, for environment attestation and verifiable code execution respectively, specific for legacy systems [56, 111]. Both schemes are vulnerable to attacks. The vulnerabilities of the former have been studied by Shankar *et al.* [114], while the vulnerabilities of the latter have been introduced in Section 4.1.

The alternative approach to software-based attestation is hardware-based attestation. The research community spent a lot of efforts in developing hardware technology equipped with special trusted components to make hardware-based attestation practical. Examples of hardware technology with such capabilities are Cerium [16], BIND [117], Intel LaGrande Technology [47], and AMD Pacifica Technology [2]. In particular, thanks to the efforts of the Trusted Computing Group and the standardization of the TPM chip [131], Intel LaGrande and AMD Pacifica technologies are slowly becoming mainstream. They have been used as ground to develop various hardware-based attestation schemes. Examples of these schemes are the IBM Integrity measurements Architecture [106], the Open Source Loader [55], Terra [41], and Flicker [78]. Similarly to Conqueror and Pioneer, Flicker's goal is to achieve tamper-proof execution of code on untrusted systems. While Conqueror and Pioneer are entirely software-based solutions, Flicker leverages the TPM, available on modern commodity hardware, to accomplish the same goal. In particular Flicker relies on a feature introduced in the CPU that allows the secure late launch of virtual machine monitors.

6.3 Dynamic analysis of commodity systems

The generic dynamic analysis infrastructure we propose in Chapter 3 shares many similarities with analysis frameworks and instrumentation techniques extensively explored in the past. In the following we briefly review some of these approaches and we compare them with the infrastructure described in Chapter 3. However, it is worth noting that, by exploiting recent facilities available on modern Intel x86 CPUs, our solution is able to combine and to offer simultaneously the main benefits introduced by previous research work.

6.3.1 Dynamic kernel instrumentation

DTrace is a facility included into the Solaris kernel that allows the dynamic instrumentation of production systems [11]. The key points of DTrace are efficiency and flexibility. First, the instrumentation framework itself introduces no overhead. Second, the framework provides tens of thousands of instrumentation points, and the actions to be taken can be expressed in terms of a high-level control language, that also includes a number of mechanisms to guarantee run-time safety. Similarly, KernInst is a dynamic instrumentation framework for commodity kernels [130]. KernInst has been developed mainly to gather information about the performances of a running kernel, but it has also been employed for run-time kernel optimization. Differently from DTrace, KernInst is not transparent to the other components of the system and does not provide any mechanism for run-time safety of the instrumentation routines. None of the aforementioned approaches is OS-independent, and they cannot be applied to closed-source operating systems. DTrace requires the pre-existence of instrumentation points in the kernel. KernInst instead requires to patch the code of the kernel of the operating system to add the missing instrumentation points. Clearly, that is possible only if low-level details about the internals of the kernel are made available. Our framework does not suffer these limitations, since it can instrument the kernel without modifying it and does not rely on any facility offered by the kernel.

6.3.2 Kernel-level debugging

Several efforts have been made to develop efficient and reliable kernel-level debuggers. Indeed, these applications are essential for many activities, such as the development of device drivers, or the analysis of malicious kernel-level modules. One of the first and most widely used kernel-level debuggers that targeted the Microsoft Windows operating system was SoftICE [119], but today the project has been discontinued. However, both commercial [127] and open-source [101] alternatives to SoftICE appeared. Modern versions of Windows already include a kernel debugging subsystem [79]. Unfortunately, to exploit the full capabilities of Microsoft's debugging infrastructure, the host being debugged must be physically linked (e.g., by means of a serial cable) with another machine. All

these approaches share a common factor: to debug kernel-level code, they leverage another kernel-level module. Obviously, that is like a dog chasing its tail. On the contrary, the generic dynamic analysis framework we propose does not require any kernel support nor to modify the kernel to add the missing support at run-time.

6.3.3 Frameworks based on virtual machines

Instead of relying on a kernel-level module to monitor other kernel code, an alternative approach consists in running the target code inside a virtual machine and performing the required analyses from the outside [42]. In [59, 140, 30] the authors propose virtual machines with execution replaying capabilities: a user can move forward and backwards through the execution history of the whole system, both for debugging and for understanding how an intrusion took place. In [18] Chow *et al.* propose Aftersight, a system that decouples execution recording from execution trace analysis, thus reducing the overhead suffered by the system where the guest operating system is run. Nowadays, Aftersight is part of the VMware platform, and other mainstream commercial products provide similar capabilities. Our analysis infrastructure can provide these functionalities even on systems not running in any virtual machine. Finally, Portokalidis *et al.* designed a solution based on execution replaying to protect smartphones from malicious threats [95]. Their idea consists in running a replica of the phone on a server: a tracer on the phone records a minimal execution trace, that is then transmitted to the server where a replica of the mobile device is run inside an emulated environment. As the server can leverage much more powerful computational resources than the smartphone, it can also perform sophisticated security analyses. This approach shares several similarities with the technique we discussed in Chapter 2.

6.3.4 Aspect-oriented programming

Aspect-orientation is a programming paradigm that promises to increase modularity by encapsulating cross-cutting concerns into separated code units, called "aspects", whose "advice" code is woven into the system automatically, by specifying the properties of the join-points. AspectC is an aspect-oriented framework that is used to customize (at compile-time) operating system kernels [24, 68, 69]. More dynamic approaches have been proposed: for example TOSKANA provides *before*, *after* and *around* advices for in-kernel functions and supports the implementation of aspects themselves as dynamically exchangeable kernel modules [31]. The framework we propose allows to achieve the same goal while being transparent and fault-tolerant.

7 Future directions

The research work presented in the previous chapters aims to overcome some of the limitations that affect current malware analysis and detection solutions. However, the techniques we described are not free from limitations. In this chapter we sketch possible improvements and extensions over the ideas we proposed, together with some directions for future work.

Multi-environment malware analysis. The dynamic framework proposed in Chapter 2 allows security labs to leverage multiple end-users' environments to increase the completeness of the analysis. Such approach opens new interesting challenges which we plan to investigate in the near future.

First, we assume that we are given a set of willing users, that we can use their systems for the analysis, and that these systems are diverse enough to trigger all the malicious behaviors of a malware. In practice we have to balance the coverage of the analysis with the number of end-users' environments available and that we are inclined to use. This is still an open problem. Second, by comparing the various execution traces collected from different end-users' environments, interesting correlations between malware behaviors and their trigger conditions could be mined. Finally, imagine that the suspicious program being monitored is found to be benign. We currently assume that, when the analysis of the process terminates, the synthetic lab environment is discarded. Consequently, every change the process could have made to the environment is lost. This is undoubtedly good for malicious processes, but probably an end-user wants to preserve the operations performed by a benign application. We are currently investigating possible solutions to this problem. As an example, when the analysis terminates, benign operations could be committed on the user's environment employing a technique similar to the one discussed in [52].

Dynamic analysis through hardware-assisted virtualization. In Chapter 3 we described a generic analysis framework that allows sophisticated and transparent analyses of both user- and system-level code. This infrastructure is

now an open-source project that is actively maintained by several developers. In the future, we plan to improve the implementation of our framework to support SMT and SMP machines, to allow the OS-aware inspection of different operating systems, and to support AMD virtualization technology [2].

As an application of our analysis infrastructure we also presented HyperDbg, a kernel-level debugger that demonstrates that our framework is very versatile and that enables new opportunities for dynamic analysis. An interesting extension of HyperDbg will be the support for kernel-level omniscient debugging. Omniscient debugging allows developers to inspect the status of their programs in past execution instants, in order to detect the cause of a failure without the need to run the target program multiple times [97]. HyperDbg could be extended to allow a user to record and inspect the values a memory location stored during the time, and the exceptions and interrupts occurred. Such a feature would ease a user to discover when a memory location of the kernel gets corrupted and which instruction is responsible for the corruption. Moreover, the ability to log asynchronous events, such as interrupts, would allow to spot defects connected to non-deterministic behaviors of the analyzed system. Our framework already offers all the necessary facilities for this kind of debugging: exception and interrupts can be traced natively by the framework and memory accesses can be traced using watchpoints.

Another interesting application of our framework will be dynamic aspect-oriented programming of operating system kernels. Several approaches have been proposed to apply AOP to kernels [24, 68, 69, 31]. The main advantage offered by our framework over the approaches proposed so far is that it does not require any modification of the source code of the kernel, nor any modification of the image in memory of the kernel. Moreover, our framework protects the running kernel from defects in the woven code. One approach to facilitate the use of such technology would be to provide programmers a source-to-source translator, to translate aspect oriented code written in languages like AspectC [23] into C code that uses the API offered by our framework. In particular, the translator would be responsible for translating pointcuts into API calls to trace the corresponding events, using advices as events handlers, and for translating all pointer dereferences into calls to inspection API to read the memory of the guest.

Software-based code attestation. Conqueror, the pure software-based attestation primitive we presented in Chapter 4, is extremely resilient against both static and dynamic attacks. We showed it is effective even when attackers are able to execute code at a very high level of privilege (e.g., at the hypervisor level). However, we believe Conqueror can still be extended in order to relax its threat model. As an example, in the future we plan to improve our solution to support even SMP and SMT systems, as modern systems typically support multiple processors or multiple execution cores. Moreover, as the checksum functions used by Conqueror are generated dynamically, obfuscated, and they exploit many subtle details of the x86 architecture, it is extremely difficult to *prove* that the primitive is actually immune to all possible attacks. Indeed, in Chapter 4 we gave only em-

pirical evidence of the effectiveness of Conqueror, but we still cannot guarantee that no attack exists that is able to defeat our attestation scheme.

Trustworthy analysis of compromised machines. We presented HyperSleuth (Chapter 5), a framework for constructing tools for the post-infection analysis of alleged compromised systems. HyperSleuth leverages the virtualization extensions provided by commodity hardware to guarantee that the results of the analyses cannot be altered, even by an attacker with kernel-level privileges. We believe HyperSleuth represents an interesting starting point to further extend in the future.

In particular, we plan to investigate the possibility to use HyperSleuth to develop a solution to bootstrap a fully-featured trustworthy environment inside an untrusted system. Imagine a user who needs to use an untrusted machine to perform security-critical operations, such as to access its bank account or to perform some e-commerce transactions. In this situation, any malicious software that runs on the untrusted machine could steal the bank credentials or the credit card details of the user. Our idea is to provide an infrastructure that can be used to take over the untrusted system, and to establish a trusted environment where standard applications (e.g., a web browsers) can be run with the guarantee that no malware can tamper with their execution, neither at the user- nor at the kernel-level. All this without the need to reboot the system. Unfortunately, HyperSleuth cannot be used as is for this purpose, because it does not support the execution of standard user-space applications on its top.

8

Conclusions

Malware is the root cause of many of the illicit activities that threaten Internet users every day. To defend against malicious software, users rely on anti-malware products to preemptively detect threats before they can infect their systems, or to remediate the damages done by the malware when infection already took place. Unfortunately, miscreants are often one step ahead of security vendors and researchers. As malware authors can study current defensive solutions, they can also try to exploit their limitations in order to make their malicious programs harder to detect and to eradicate. In this dissertation, we addressed the problems that affect today's security technologies, and we provided novel solutions to improve the effectiveness of state-of-the-art anti-malware products.

As a first contribution, we presented a framework that enables sophisticated behavior-based analyses of suspicious programs in multiple realistic and heterogeneous environments. We achieve this goal by distributing the execution of the program between the security laboratory (with virtually unlimited computational resources) and the environments of potential victims (which are heterogeneous by definition and might affect differently the behavior of the analyzed program), by forwarding to the latter certain system calls. We have implemented an experimental prototype to validate our idea and integrated it into an existing behavior-based malware detector. Our evaluation demonstrated the feasibility of the proposed approach, that the overhead introduced is very small, and that the analysis of multiple execution traces of the same malware sample in multiple end-users' environments can improve the results of the analysis.

In the dissertation we also proposed an infrastructure to perform complex run-time analyses of both user- and system-level code on commodity production systems. The framework exposes an API that eases the development of analysis tools on its top. The approach we described leverages hardware extensions for virtualization available on modern processors to overcome the limitations that affect existing approaches for the analysis of system-level code. In particular, the solution we suggested is transparent, does not require to recompile or reboot the

target system, it is almost completely OS-independent, and it guarantees that a defect in an analysis tool cannot damage the framework itself nor the analyzed system. Such an infrastructure is extremely valuable to analyze malware that includes kernel-level components. To demonstrate its potentials, we developed HyperDbg, an interactive kernel-level debugger for Microsoft Windows XP and Linux. HyperDbg and the whole framework have been released as an open source package.

Unfortunately, if the target system has already been compromised, the solutions we just described are completely ineffective, as a malware could tamper with the execution or the installation of the aforementioned frameworks, and hide its presence on the machine. We addressed this problem with Conqueror, a software-based code attestation scheme for tamper-proof code execution on untrusted legacy systems. Conqueror allows to execute an arbitrary piece of code with the guarantee that it is run untampered, even when no specific hardware for trusted computing is available. We developed an experimental prototype of Conqueror, to evaluate its resilience against hypervisor-based attacks, the most powerful type of dynamic attack, and against attacks based on static analysis of the code.

Finally, we introduced HyperSleuth, a framework for the analysis of alleged compromised systems. HyperSleuth guarantees that the results of the analyses cannot be altered, even by an attacker with kernel-level privileges. HyperSleuth leverages our virtualization-based analysis framework to install itself on a potentially compromised system as it runs. The installation of the hypervisor is attested by Conqueror, our secure loader. We developed a proof-of-concept prototype of HyperSleuth and, on top of it, we implemented three forensic analysis applications: a lazy physical memory dumper, a lie detector, and a system call tracer. These applications can be used to take over a potentially compromised host and to securely collect the data necessary to verify the presence of malicious code. The experimental evaluation testified the effectiveness of the proposed approach.

Malicious threats are continuously evolving subjects, and their authors invest more and more efforts trying to improve their creations. Today's interconnected world provides a plethora of technologies that malware authors could exploit in order to create new threats. SCADA networks, VoIP infrastructures, mobile devices, and modern web technologies are just some examples of the possible targets of tomorrow's malware. At the same time, anti-malware manufacturers develop defensive countermeasures that can barely keep up with malicious technology. But how to break this arms race between miscreants and the anti-malware community?

During the last years, the academic research made the headway on innovative security solutions. However, both marketing and technical issues prevent a wide adoption of new defensive approaches. As an example, the majority of today's deployed anti-virus software is one step behind malware: many hosts still leverage

signature-based products, as these approaches continue to be more reliable and more scalable than behavior-based technologies. In our opinion, it is essential that the research community starts to address some of the practical challenges that prevent the wide adoption of innovative defensive solutions (e.g., how to perform sophisticated analyses at the end host? How to reduce false positives?). It is fundamental to invest even more efforts into these topics in order to keep up with the evolving malware landscape.

Secondly, proactive approaches remain the only effective protection against malicious threats. After a malware sample infects a host, it is really difficult to provide a post-infection solution that can remediate the infection and clean up the compromised machine. In this dissertation we described different techniques to support the execution of post-infection tools. Nevertheless, little research focuses on the development of remediation approaches that can be applied on end-users' machines. In our previous work we made a first step towards the automatic generation of remediation procedures [88]. However, there are still many challenges that need to be addressed to build effective post-infection solutions, with the final intent to provide end-users' defensive software that is one step ahead malicious threats.

Bibliography

[1] Keith Adams. Blue Pill detection in two easy steps, 2007.

[2] AMD, Inc. AMD virtualization. (http://www.amd.com/virtualization).

[3] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient detection of split personalities in malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, February 2010.

[4] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 25th Symposium On Applied Computing (SAC)*, Lusanne, Switzerland, March 2010.

[5] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A tool for analyzing malware. In *Proceedings of the Annual Conference of the European Institute for Computer Antivirus Research*, 2006.

[6] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones and Bartlett Publishers, Inc., USA, 2009.

[7] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. BitScope: Automatically dissecting malicious binaries. Technical Report CMU-CS-07-133, Carnegie Mellon University, March 2007.

[8] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University, 2007.

[9] Jamie Butler and Peter Silberman. RAIDE: Rookit analysis identification elimination. In *Black Hat USA*, 2006.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (SIGSAC)*. ACM SIGSAC, 2006.

[11] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX Annual Technical Conference*, June 2004.

[12] Martim Carbone, Diego Zamboni, and Wenke Lee. Taming virtualization. *IEEE Security and Privacy*, 6(1), 2008.

[13] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[14] Cendio. SeamlessRDP – Seamless windows support for rdesktop. (http://www.cendio.com/seamlessrdp/).

[15] Milind Chabbi. Efficient taint analysis using multicore machines. Master's thesis, University of Arizona, 2007.

[16] Benjie Chen and Robert Morris. Certifying program execution with secure processors. In *Proceedings of the 9th conference on Hot Topics in Operating Systems*, 2003.

[17] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *Operating Systems Review*, 42(2), 2008.

[18] Jim Chow, Tal Garfinkel, and Peter Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of USENIX Annual Technical Conference*, June 2008.

[19] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium*. USENIX Association, 2003.

[20] Mihai Christodorescu and Somesh Jha. Testing malware detectors. *SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004.

[21] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceeding of the 1st Annual India Software Engineering Conference (ISEC)*, Hyderabad, India, February 2008.

[22] Daniele Perito Claude Castelluccia, Aurélien Francillon and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*, 2009.

[23] Yvonne Coady, Gregor Kiczales, Michael J. Feeley, Norman C. Hutchinson, and Joon Suan Ong. Structuring operating system aspects. *Communications of the ACM*, 44(10):79–82, 2001.

[24] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering Conference*, 2001.

[25] Fred Cohen. Computer viruses, theory and experiments. *Computers & Security*, 6, 1987.

[26] Cyveillance. Malware detection rates for leading AV solutions. Technical report, Cyveillance, August 2010.

[27] Dino Dai Zovi. Hardware virtualization based rootkits. Black Hat USA, 2006.

[28] Dancho Danchev. Conficker's estimated economic cost? $9.1 billion.

[29] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[30] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementations (OSDI)*, December 2002.

[31] Michael Engel and Bernd Freisleben. TOSKANA: A toolkit for operating system kernel aspects. *Transactions on Aspect-Oriented Software Development II*, 4242:182–226, 2006.

[32] F-Secure. Trojan information pages: Bancos.VE. (http://www.f-secure.com/v-descs/bancos_ve.shtml).

[33] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, Antwerp, Belgium, September 2010.

[34] Peter Ferrie, Nate Lawson, and Thomas Ptacek. Don't tell Joanna, the virtualized rootkit is dead. Black Hat USA, 2007.

[35] Stephanie Forrest, Steven R. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 1996.

[36] Jason Franklin, Adrian Perrig, Vern Paxson, and Stefan Savage. An inquiry into the nature and causes of the wealth of Internet miscreants. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, pages 375–388. ACM, 2007.

[37] Jason Franklin, Arvind Seshadri, Ning Qu, Anupam Datta, and Sagar Chaki. Attacking, repairing, and verifying SecVisor: A retrospective on the security of a hypervisor. Technical report, Carnegie Mellon University, 2008.

[38] Merrick Furst. Expert: Botnets no. 1 emerging internet threat. *CNN Technology*, 2006.

[39] Juan A. Garay and Lorenz Huelsbergen. Software integrity protection using timed executable agents. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security (ASIACCS)*, 2006.

[40] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, 2007.

[41] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM symposium on Operating systems principles*, 2003.

[42] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, San Diego, CA, USA, February 2003.

[43] Jonathon Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[44] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *ACM SIGPLAN Notices*, 40(6), June 2005.

[45] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[46] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the USENIX Security Symposium*, 1996.

[47] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 2009.

[48] Peter Gutmann. The commercial malware industry, 2007.

[49] Hex-Rays. IDA Pro. (http://www.hex-rays.com/idapro/).

[50] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the EuroSys Conference*, 2006.

[51] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2006.

[52] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[53] Intel, Inc. Intel virtualization technology. (http://www.intel.com/technology/virtualization/).

[54] Xuxian Jiang and Xinyuan Wang. "Out-of-the-Box" monitoring of VM-based high-interaction honeypots. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[55] Bernhard Kauer. OSLO: Improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[56] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[57] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[58] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2006.

[59] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of USENIX Annual Technical Conference*, April 2005.

[60] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[61] Alexander Klimov and Adi Shamir. A new class of invertible mappings. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, 2003.

[62] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[63] Andreas Moser Christopher Kruegel and Engin Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[64] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), 1992.

[65] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A smart fuzzer for x86 executables. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems (SESS)*, Minneapolis, MN, USA, May 2007.

[66] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS)*, 2003.

[67] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP Journal*, 1997.

[68] Daniel Mahrenholz, Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. An aspect-oriented implementation of interrupt synchronization in the PURE operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, June 2002.

[69] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Program instrumentation for debugging and monitoring with AspectC++. In *Proceedings of the Symposium on Object-Oriented Real-Time Distributed Computing*, April 2002.

[70] John Markoff. Attack of the zombie computers is a growing threat, experts say. *The New York Times*, January 2007.

[71] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Ottawa, Canada, September 2010.

[72] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. A framework for behavior-based malware analysis in the cloud. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, Kolkata, India, December 2009. Springer.

[73] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, Bonn, Germany, July 2010. Springer.

[74] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA)*, pages 261–272, Chicago, Illinois, USA, July 2009. ACM.

[75] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *Proceedings of the 2010 International Symposium on Testing and Analysis (ISSTA)*, Trento, Italy, July 2010.

[76] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[77] McAfee, Inc. Operation Aurora. (http://www.mcafee.com/us/threat_center/operation_aurora.html).

[78] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, 2008.

[79] Microsoft Corporation. Debugging tools for Windows.

[80] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceeding of the 2007 IEEE Symposium on Security and Privacy (Oakland)*, 2007.

[81] Carey Nachenberg. Understanding and managing polymorphic viruses. Technical report, Symantec, Inc., September 1996.

[82] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[83] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[84] NoAH Consortium. Containment environment design. Technical report, European Network of Affined Honeypots, 2006.

[85] NovaShield. (http://www.novashield.com/).

[86] Jon Oberheide, Evan Cooke, and Farnam Jahanian. CloudAV: N-Version antivirus in the network cloud. In *Proceedings of the USENIX Security Symposium*, 2008.

[87] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3^{rd} USENIX Workshop on Offensive Technologies (WOOT)*, Montreal, Canada, August 2009. ACM.

[88] Roberto Paleari, Lorenzo Martignoni, Emanuele Passerini, Drew Davidson, Matt Fredrikson, Jon Giffin, and Somesh Jha. Automatic generation of remediation procedures for malware infections. In *Proceedings of the 19^{th} USENIX Security Symposium*, Washington, DC, USA, August 2010.

[89] Panda Security. True Prevent. (http://research.pandasecurity.com/archive/How-TruPrevent-Works-_2800_I_2900_.aspx).

[90] Panda Security. From traditional antivirus to collective intelligence, 2007.

[91] Emanuele Passerini, Roberto Paleari, and Lorenzo Martignoni. How good are malware detectors at remediating infected systems? In *Proceedings of the 6^{th} Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, Como, Italy, July 2009. Springer.

[92] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[93] Adrian Perrig, Virgil Gligor, and Amit Vasudevan. XTREC: secure real-time execution trace recording and analysis on commodity platforms. Technical report, Carnegie Mellon University, 2010.

[94] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An analysis of Conficker's logic and rendezvous points. Technical report, SRI International, 2009.

[95] Georgios Portokalidis, Philip Homburg, and Herbert Bos. Paranoid android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*. IEEE, December 2010.

[96] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, Leuven, Belgium, April 2006.

[97] Guillaume Pothier and Eric Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26:78–85, 2009.

[98] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems*, 30(5), August 2008.

[99] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[100] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[101] Rasta ring 0 debugger. (http://rr0d.droids-corp.org/).

[102] Mark Russinovich and David Solomon. *Microsoft Windows Internals*. Microsoft Press, 4th edition, 2004.

[103] Joanna Rutkowska. Subverting vista kernel for fun and profit. Black Hat USA.

[104] Joanna Rutkowska and Alexander Tereshkin. IsGameOver() anyone? Black Hat USA, 2007.

[105] Ravi Sahita, Ulhas Warrier, and Prashant Dewan. Dynamic software application protection. Technical report, Intel Corporation, 2009.

[106] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[107] Sana Security. (http://www.sanasecurity.com/).

[108] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software attestation for key establishment in sensor networks. In *Proceedings of the 2008 International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2008.

[109] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the ACM Workshop on Wireless Security (WiSe)*, 2006.

[110] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, 2007.

[111] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[112] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2004.

[113] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In *Security and Privacy in Ad-hoc and Sensor Networks*, 2005.

[114] Umesh Shankar, Monica Chew, and J.D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13^{th} USENIX Security Symposium*, 2004.

[115] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[116] Monirul Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[117] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland)*, 2005.

[118] Jim E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[119] SoftICE. (http://en.wikipedia.org/wiki/SoftICE).

[120] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (Keynote invited paper)*, Hyderabad, India, December 2008.

[121] Eugene H. Spafford. The Internet Worm program: An analysis. *Computer Communications*, 19(1):17–57, January 1989.

[122] Sherri Sparks and Jamie Butler. Shadow Walker. raising the bar for windows rootkit detection. Phrack Magazine (Vol. 11, No. 63), 2005.

[123] Adrian Stepan. Improving proactive detection of packed malware. *Virus Bulletin*, March 2006.

[124] Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2005.

[125] Sun Microsystems, Inc. Sun xVM VirtualBox. (http://www.virtualbox.org/).

[126] Symantec, Inc. Symantec global internet security threat report: Volume XV. Technical report, Symantec, Inc., April 2010.

[127] Syser kernel debugger. (http://www.sysersoft.com/).

[128] Peter Ször. Hunting for metamorphic. Technical report, Symantec, Inc., June 2003.

[129] Peter Ször. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, 2005.

[130] Ariel Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. PhD thesis, University of Wisconsin-Madison, 2001.

[131] Trusted Computing Group. (http://www.trustedcomputinggroup.org/).

[132] Amit Vasudevan and Ramesh Yerraballi. Stealth breakpoints. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[133] VirtualSquare. Remote system call. (http://wiki.virtualsquare.org/index.php/Remote_System_Call).

[134] VMPsoft. VMProtect. (http://www.vmprotect.ru/).

[135] Volatile Systems LLC. Volatility. (http://www.volatilesystems.com/).

[136] Wikipedia, the free encyclopedia. Computer forensics. (http://en.wikipedia.org/wiki/Computer_forensics).

[137] Jeffrey Wilhelm and Tzi cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Gold Goast, Australia, 2007.

[138] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 2007.

[139] Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland)*, 2005.

[140] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[141] Heng Yin, Dawn Song, Manuel Egele, Engin Kirda, and Christopher Kruegel. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2007.