FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

Data-only Malware

Sebastian Wolfgang Vogl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 09.02.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.07.2015 angenommen.

Acknowledgements

Over the past years, I have received support and encouragement from many smart and amazing people. I want to seize this opportunity to express my sincere appreciation and gratitude to all of them.

First and foremost, I would like to extend my thanks to my advisor and supervisor, Prof. Dr. Claudia Eckert, for providing me with the opportunity to write this thesis and for her outstanding mentoring during this time. Her unwavering support, continuous encouragement, and guidance greatly helped my research and this dissertation.

Similarly, I want to thank my second advisor, Prof. Dr. Thorsten Holz, for his assistance, valuable advice, and crucial contribution to my research, which advanced and improved my thesis substantially.

Additionally, I am very grateful to Prof. Dr. Michael Gerndt and Prof. Dr. Jonathon Giffin for providing me with the possibility of studying at Georgia Tech and for their support, encouragement, and guidance throughout the process.

Next, I would like to thank my former and current colleagues at the IT security research groups in Munich and Bochum for the interesting discussions, the collaboration, the support of my work, the excellent atmosphere, and the pleasant evenings: Dr. Christian Schneider, Dr. Jonas Pfoh, Thomas Kittel, George Webster, Tamas Lengyel, Fatih Kilic, Julian Kirsch, Robert Gawlik, and Behrad Garmany. I am also grateful to the great people I met during my studies in Munich and in Atlanta for making studying an unforgettable experience and expanding my horizon. Most importantly, Tobias Röhm, Sepp Tremmel, Markus Graßl, Ferdinand Beyer, Felix Weninger, Bulli Bertolotti, Paolo Manenti, and Peter Ligeiro. My thanks also go to the extraordinary students that contributed to my projects: Lorenz Panny, Christian von Pentz, and Jonas Jelten.

Thanks, too, to my closest friends for always being there for me, for their general awesomeness, and for helping me to keep my sanity: Felix Römisch, Thomas Zirngibl, Philip Lembcke, Alexander Lehmann, Melanie Lehmann, Felix Abele, and Dominik Zaun.

Finally yet importantly, I would like to thank my family. My parents who opened up this path for me with their unlimited support, encouragement, and love. My sister for always looking out for her little brother. My grandparents for enriching my life, introducing me to Bud Spencer, and helping me to put things in perspective. Ronnie for his meticulous help with grammar and text comprehension. Family Meier for their support, understanding, and encouragement. Anna and Markus for becoming part of my life. And my love, Elisabeth, for being you, which is more than I ever dreamed of.

Abstract

Protecting the integrity of code is generally considered as one of the most effective approaches to counteract malicious software (malware). However, the fundamental problem with code-based detection approaches is that they rely on the false assumption that all malware consists of executable instructions. This makes them vulnerable to *data-only* malware, which, in contrast to traditional malware, does not introduce any additional instructions into the infected system. Instead, this malware form solely relies on the instructions that existed before its presence to perform malicious computations. For this purpose, data-only malware employs code reuse techniques such as return-oriented programming to combine existing instructions into a new malicious program. Due to this approach, the malware itself will consist solely of control data, enabling it to evade all existing code-based detection mechanisms. Despite this astonishing capability and the obvious risks associated with it, data-only malware has not been studied in detail to date. For this reason, the dimensions of the danger of this potential future threat remain as yet unknown.

To remedy this shortcoming, we will in this work provide the first comprehensive study of data-only malware. We will begin by conducting a detailed analysis of data-only malware to determine the capabilities and limitations of this new malware form. In the process, we will show that data-only malware is not only on a par with traditional malware, but even surpasses it in its level of stealth and its ability to evade detection. To demonstrate this, we will present detailed proof of concept implementations of sophisticated data-only malware that are capable of infecting current systems in spite of the numerous protection mechanisms that they at present employ.

Having shown that data-only malware is a serious and realistic threat, we evaluate the effectiveness of existing defense mechanisms with regard to data-only malware in the second part of this thesis. The goal of our analysis is hereby to determine whether there already exist effective countermeasures against data-only malware or if this new malware form poses an immediate danger to current systems due to the lack of such. In the course of our analysis, we identify hook-based detection mechanisms as the only potentially effective existing countermeasure against data-only malware. To validate this hypothesis, we follow our initial analysis with a detailed study of current hook-based detection mechanisms. In the process, we discover that hook-based detection mechanisms rely on the false assumption that an attacker can only modify persistent control data in order to install hooks. This oversight enables data-only malware to evade existing mechanisms by targeting transient control data such as return addresses instead. To illustrate this, we present a new hooking concept that we refer to as dynamic hooking. Instead of changing control data directly, the key idea behind this concept is to manipulate non-control data in such a way that it will trigger a vulnerability at runtime, which then overwrites transient control data, resulting in the invocation of the hook. Due to this approach, dynamic hooks are hidden within non-control data, which makes it significantly more difficult to detect them and enables them to evade all existing hook-based detection mechanisms.

Since our analysis of existing malware defense mechanisms yielded the result that even hook-based defense mechanisms are unable to detect data-only malware, we will deal with countermeasures against this malware form in the third and final part of the thesis. For this purpose, we first introduce a virtual machine introspection-based framework for malware detection and removal called X-TIER. X-TIER enables security applications to inject kernel modules from the hypervisor into a running virtual machine and to execute them securely within the guest. In the process, the modules can access any kernel function and any kernel data structure without loss of security. In addition, the modules can transfer arbitrary information to the hypervisor. Consequently, X-TIER effectively enables hypervisor-based security applications to circumvent the semantic gap, which constitutes the key problem that all security applications on the hypervisor-level face. By combining strong security guarantees with full access to the state of the virtual machine, our framework can provide an excellent basis for countermeasures against data-only malware.

Based on our framework we finally present three concrete detection mechanisms for data-only malware. Each of these mechanisms puts to use one of the inherent dependencies of data-only malware, which we identified during our initial analysis of this malware form, against the malware itself. This results in effective countermeasures that can, particularly when used in combination, provide strong initial defenses against data-only malware.

Zusammenfassung

Die Integrität des Systemcodes zu schützen, gilt allgemein als eine der effektivsten Methoden um Infektionen durch Schadsoftware zu verhindern. Das fundamentale Problem solcher codebasierten Erkennungsmethoden ist jedoch, dass sie auf der falschen Annahme basieren, dass jede Schadsoftware aus ausführbaren Maschineninstruktionen besteht. Dadurch sind derartige Erkennungsmechanismen anfällig für *rein datenbasierte* Schadsoftware, die im Gegensatz zu traditioneller Schadsoftware keine zusätzlichen Instruktionen in das System einschleust. Stattdessen, verwendet diese Schadsoftwareart zur Ausführung ausschließlich Instruktionen, die sich bereits vor der Infektion auf dem System befunden haben. Dazu fügt die rein datenbasierte Schadsoftware bestehende Instruktionen mit Hilfe sogenannter Code-Reuse-Techniken wie Return-Oriented Programming zu einem neuen Schadprogramm zusammen. Die resultierende Schadsoftware besteht dabei ausschließlich aus Kontrolldaten, was es ihr ermöglicht allen existierenden codebasierten Erkennungsverfahren zu entgehen. Trotz dieser erstaunlichen Fähigkeit und dem damit verbundenem Risiko, wurde rein datenbasierte Schadsoftware in der Forschung bisher nur unzureichend betrachtet. Aus diesem Grund ist derzeit völlig unklar, wie groß die Gefahr ist, die von dieser zukünftigen Bedrohung ausgeht.

Um diesen Missstand zu beseitigen, führen wir in dieser Arbeit die erste vollständige Untersuchung von rein datenbasierter Schadsoftware durch. Dazu werden wir im ersten Teil der Arbeit eine detaillierte Analyse rein datenbasierter Schadsoftware vornehmen und die Fähigkeiten und Limitationen dieser Schadsoftwareform ermitteln. Wir zeigen, dass rein datenbasierte Schadsoftware traditioneller Schadsoftware in vielen Bereichen ebenbürtig und was ihre Tarnungsfähigkeit und ihre Entdeckunsgvermeidung anbelangt sogar überlegen ist. Um diese Tatsache zu untermauern, präsentieren wir detaillierte Beispielimplementierung von komplexer rein datenbasierter Schadsoftware, die trotz aller heutigen Schutzmechanismen in der Lage ist aktuelle Systeme zu infizieren.

Nachdem wir gezeigt haben, dass rein datenbasierte Schadsoftware eine reale und ernst zu nehmende Bedrohung darstellt, evaluieren wir im zweiten Teil der Arbeit die Effektivität heutiger Erkennungsmechanismen in Bezug auf diese neue Schadsoftwareform. Das Ziel unserer Analyse ist dabei herauszufinden, ob es bereits einen effektiven Schutzmechanismus gegen rein datenbasierte Schadsoftware gibt oder ob diese neue Schadsoftwareart eine akute Gefahr für heutige Systeme darstellt. In unserer Analyse stellen wir fest, dass derzeit nur hookbasierte Erkennungsmechanismen in der Lage zu sein scheinen rein datenbasierte Schadsoftware zu erkennen. Um diese Hypothese zu überprüfen, führen wir eine detaillierte Untersuchung dieser Erkennungsmethoden durch. Dabei zeigt sich, dass derzeitige hookbasierte Erkennungsverfahren auf der falschen Annahme beruhen, dass Hooks nur in persistenten Kontrolldaten platziert werden können. Das führt dazu, dass rein datenbasierte Schadsoftware den bestehenden Verfahren entgehen kann, indem sie sich transiente Kontrolldaten wie Rücksprungadressen zu Nutze macht. Um dies zu zeigen, schlagen wir ein neues Hookingkonzept vor, das wir dynamisches Hooking nennen. Anstatt Kontrolldaten direkt zu verändern, werden dabei normale Daten so manipuliert, dass sie Schwachstellen aktivieren, die transiente Kontrolldaten zur Laufzeit überschreiben. Dadurch wird der eigentliche Hook in normalen Daten versteckt, was die Erkennung dynamischer Hooks deutlich schwieriger macht und es dieser Hookform ermöglicht existierenden Erkennungsverfahren zu entgehen.

Da unsere Analyse der bestehenden Erkennungsmethoden ergeben hat, dass selbst hookbasierte Verfahren nicht in der Lage sind rein datenbasierte Schadsoftware zu erkennen, beschäftigen wir uns im dritten und letzten Teil der Arbeit mit Gegenmaßnahme für diese neue Schadsoftwareart. Dazu präsentieren wir zunächst ein hypervisorbasiertes Framework für die Erkennung und Beseitigung von Schadsoftware namens X-TIER. Durch X-TIER erhalten Sicherheitsapplikationen die Möglichkeit Kernelmodule vom Hypervisor in eine laufende virtuelle Maschine zu laden und diese dort sicher auszuführen. Die injizierten Module können dabei auf jede beliebige Kernelfunktion und Kerneldatenstruktur zuzugreifen ohne ihre Sicherheit zu gefährden. Zusätzlich können die vom Modul erhaltenen Informationen zum Hypervisor transferiert werden. Dies ermöglicht Sicherheitsapplikationen die semantische Lücke, ein zentrales Problem aller hypervisorbasierten Applikationen, auf Hypervisorebene zu umgehen. Da unser Framework hohe Sicherheit mit dem vollständigen Zugriff auf die virtuelle Maschine verbindet, bietet es eine exzellente Basis um rein datenbasierter Schadsoftware entgegenzuwirken.

Auf unserem Framework aufbauend präsentieren wir schließlich drei konkrete Erkennungsmechanismen für rein datenbasierte Schadsoftware. Jeder dieser Mechanismen macht sich dabei eine der während unserer initialen Analyse identifizierten inhärenten Abhängigkeiten rein datenbasierter Schadsoftware zu Nutze. Dies resultiert in effizienten Gegenmaßnahmen, die insbesondere in Kombination, einen starken initialen Schutz gegen rein datenbasierte Schadsoftware bieten.

Contents

List of Publications

[1] **Sebastian Vogl** and Claudia Eckert. "Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture". In: *Proceedings of EuroSec'12, 5th European Workshop on System Security,* ACM Press, 2012.

[2] **Sebastian Vogl**, Fatih Kilic, Christian Schneider, and Claudia Eckert. "X-TIER: Kernel Module Injection". In: *Proceedings of the 7th International Conference on Network and System Security.* Vol. 7873. Lecture Notes in Computer Science. Springer, June 2013, pp. 192–206.

[3] **Sebastian Vogl**, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. "Persistent Data-only Malware: Function Hooks without Code". In: *Proceedings of the 21th Annual Network & Distributed System Security Symposium (NDSS).* Feb. 2014.

[4] **Sebastian Vogl**, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. "Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data". In: *Proceedings of the 23rd USENIX Security Symposium.* USENIX, Aug. 2014.

[5] Thomas Kittel, **Sebastian Vogl**, Tamas K. Lengyel, Jonas Pfoh, and Claudia Eckert. "Code Validation for Modern OS Kernels". In: *1st Workshop on Malware Memory Forensics (MMF).* Dec. 2014.

[6] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, **Sebastian Vogl**, and Aggelos Kiayias. "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System". In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC).* Dec. 2014.

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Malicious software (malware) is without doubt one of the biggest computer security threats today. While malware has been an issue ever since the first virus appeared in the 1970s [70], the problem has become worse in the course of the last ten years to such an extent that it is now on the verge of getting out of control. To verify this, we only have to take a look at recent threat reports. For example, Check Point [121] estimated in their security report for 2014 that malware is downloaded to a computer in a corporate environment *every* 10 minutes on average. As a result, a host within a corporate environment becomes infected *every* 24 hours on average in spite of security mechanisms such as antivirus software and firewalls. Symantec [32] discovered in their newest Internet threat report that one e-mail and one website in every 196 e-mails and 566 websites contains malware. Pandalabs [106] observed more than 160,000 *new* malware samples *every* day in their first quarterly report 2014. They further estimated the *global* infection rate of machines to be 32.77%. Almost a third of all machines on the planet are therefore infected with malware.

One of the main reasons for this development is that the *motivation* behind the creation of malware has changed drastically in the course of the last ten years. Up to the early 2000s, malware was primarily developed by individuals for research purposes or merely as a prank. The majority of malware had no financial motivation. Instead, malware could be seen as a form of "computer vandalism" [70]. This is amply demonstrated by the instances of malware that appeared during this period of time. Consider the *Blaster* worm [6], for example, which infected more than 100,000 systems in the year 2003. In contrast to the malware that we see today, it operated in user space with very little in the way of defensive mechanisms and only contained a single malicious functionality: to launch a distributed denial of service (DDOS) attack against "windowsupdate.com". Such a low level of sophistication paired with destructive behavior is typical of malware at that time.

In the following years, however, malware was discovered as a *business model* for cyber criminals, as a tool for *cyber espionage*, and as a weapon in *cyber warfare* [70]. As a result, the focus of malware developers shifted from destruction to achieving financial, political, or military objectives. Avoiding detection suddenly became a crucial aspect of malware creation, because the longer the malware remains hidden, the higher the revenue for its creators. As a consequence, the sophistication level of malware increased rapidly. Instead of simple user space malware, we now see complex kernel space malware such as Stuxnet, Duqu, and Flame [9] that show an increase in sophistication in the target of their attack, the exploitation methods used to deliver them, and their ability to evade detection. This makes the detection of malware increasingly difficult, especially as malware developers continue to research novel attack vectors, enabling them always to stay one-step ahead of the defenders.

1.1 Motivation

The Current Anti-Malware Situation. Malware is constantly evolving. As a result, thousands of new malware variants appear every day [106]. For instance, Check Point's security report for 2014 estimated that a computer within a corporate environment downloads an *unknown* malware sample every 27 minutes on average [121]. The same report states that less than 10% of the available antivirus engines are *able* to detect unknown malware. If we consider these numbers, it is no wonder that malware has infected a third of the machines on the Internet. In fact, in light of these results, even antivirus vendors such as Symantec go so far as to say that the antivirus in its current form is essentially "dead" given the fact that it is unable to stop most unknown attacks [132].

The problem arises because antivirus vendors still primarily rely on signatures for detection. That is, they merely *react* to malware infections. As a consequence, unknown malware can spread until it has been analyzed and signatures have been created for it. Naturally, this process takes quite some time. A study by Lastline Labs [166] found, for example, that it takes in general two weeks till the detection rate for previously unknown malware samples significantly improves. Given the number of new malware samples appearing every day, the authors thus conclude that "AV [antivirus] isn't dead, it just can't keep up" [166].

Being Proactive instead of Reactive. To solve this problem we have to move away from *reactive* malware detection towards *proactive* malware defense. Instead of waiting for malware developers to create ever more sophisticated malware, we have to identify and analyze possible future threats ourselves in order to mitigate attack vectors *before* they can be exploited. However, to pursue this approach it is necessary to research novel *attacks* in addition to defense mechanisms. By slipping into the role of the attacker, we can detect weaknesses in our current defenses, which will allow us to identify future malware threats more reliably. Once identified, we can study and analyze a threat to obtain a profound understanding of it. This will not only allow us to assess its risk accurately, but it will also enable us to design effective countermeasures against it before it becomes a reality. A critical threat that is just beyond the horizon and possibly represents the next step in malware evolution is **data-only malware** [62, 170].

Data-only Malware. The key property that distinguishes data-only malware from existing malware forms is that it does not introduce any *new* instructions into the system it infects. Instead, this malware form utilizes the instructions that already existed *before* its presence to implement its own functionality. For this purpose, data-only malware makes use of *code reuse techniques* such as return-oriented programming (ROP)[1] to combine existing instruction sequences into a new program. This enables data-only malware to circumvent one of the most crucial constraints that malware has faced so far: it is capable of infecting a system *without* changing its codebase.

Code-based Detection. Up until now, one of the biggest weaknesses of malware has been that it has always consisted of executable instructions. Consequently, no matter how sophisticated malware was, it had to change the current *codebase* of a system in order to run by either introducing new code or modifying existing instructions. Since this requirement was inherent to all malware independent of its type, this property was naturally predestined to be used in the detection of malware and thus formed the core of many recent protection mechanisms that were supposed to improve the current malware detection rate. Three prominent real world examples that reflect this development are PatchGuard, secure boot, and supervisior mode execution prevention (SMEP), which all try to protect the codebase of a system in order to thwart malware infections. In addition, researchers have proposed a variety of mechanisms to validate the integrity of code at runtime. One of the most sophisticated approaches in this direction is currently Patagonix [87], which is capable of validating the integrity of code regions of arbitrary binaries (including the operating system (OS) kernel) in memory. While this is, by no means, an exhaustive list, it illustrates the fact that protecting against the introduction or modification of code is generally considered an effective method of counteracting malware.

Future Threat? Data-only malware was specifically designed to evade code-based detection approaches. While this alone is a strong argument in favor of the thesis that data-only malware is a likely next step in malware evolution, it is not the only argument that supports this theory. Another important indicator is the field of exploitation, which is naturally strongly intertwined with the area of malware creation and has undergone a very similar evolution. While exploits formerly used code to achieve their goals, this approach is in the meanwhile no longer possible owing to the introduction of the aforementioned protection mechanisms. As a consequence, exploits nowadays primarily rely on code reuse techniques. Malware is currently facing exactly the same problem. With the deployment of more and more code-based detection mechanisms, it will become

[1]We cover ROP as well as other code reuse techniques in more detail in Section 2.2.1.2.

increasingly harder for malware authors to implement traditional malware. At some point in the future, malware authors will thus be forced to find an alternative to code-based malware. Since code reuse techniques have proved to be an effective solution in the case of exploitation, it is therefore probable that malware authors will, like exploit writers, turn towards this technique and accordingly to data-only malware. In spite of these facts, data-only malware has not been studied in detail to date and thus many of its capabilities and properties remain unknown.

Summary. In this thesis, we want to explore the capabilities and limitations of data-only malware in depth so that we can obtain a profound understanding of this possible future threat. The goal of our research is thereby, on the one hand, to assess the threat that data-only malware poses for current systems, a threat which remains, due to lack of research in the field, for the most part obscure, and, on the other hand, to propose initial defense mechanisms against this threat, should it prove dangerous. Our motivation for this work is twofold. First, as stated above, current antivirus software still primarily relies on reactive mechanisms for malware protection. However, this approach leaves current systems vulnerable to unknown attacks such as data-only malware, which, as we saw from the current threat reports, are increasing in number all the time. To solve this problem, we must move away from solely conducting defensive research towards conducting offensive research as well. By putting ourselves in the shoes of the attacker, we obtain the opportunity of identifying and analyzing future threats at an early stage. This will enable us to mitigate threats *before* they are exploited, which will significantly increase the security of current systems.

Second, given this line of reasoning, the question arises as to what constitutes a likely future threat that is worth studying. To answer this question, we briefly looked at current defense mechanisms and established that code-based malware detection represents a very promising approach in countering the threat of traditional malware as it uses one of the fundamental properties of current malware in the process. Since malware developers will naturally try to evade this detection approach, they will search for alternative ways to implement malware. Similar to the case of exploitation, it is therefore highly likely that they will turn towards implementing malware solely by using code reuse techniques. That is, they will turn towards data-only malware. In spite of this fact, existing knowledge about data-only malware can currently be described as at best rudimentary.

1.2 Research Questions

This thesis aims to address two central research questions: how much of a threat does data-only malware pose to the security of current systems? And immediately following from this question: how can we mitigate the threat should it prove dangerous? These questions can be further divided into three concrete subproblems:

(Q1) What are the capabilities and limitations of data-only malware in practice? Since data-only malware is a novel threat that has only been researched in rudimentary fashion so far, the initial question that must be answered is what are the actual capabilities and limitations of this malware form. In this connection it is important to state that data-only malware relies on code reuse to function. Consequently, the computational ability of data-only malware always depends on the instructions that the target system provides. Considering this constraint, the fundamental question arises whether data-only malware can provide the same functionality and properties as traditional malware. In other words, is data-only malware as *powerful* as existing malware forms and thus an equally *realistic* threat?

(Q2) How effective are current countermeasures against data-only malware? Having determined what data-only malware is capable of, the second question that arises is how well are current systems protected against this novel threat. Clearly, if existing defense mechanisms were already able to hinder data-only malware, the threat that the malware form poses would be relatively small. Consequently, we have to analyze the effectiveness of existing protection mechanisms against data-only malware. In this context, we must also consider the *evasion resistance* of potential defense mechanisms to ensure that an attacker could not simply alter her methods in order to bypass them.

(Q3) How can we protect current systems from data-only malware? From the previous questions there naturally follows a third: how can we counteract the threat of data-only malware should it prove realistic and dangerous? To address this problem, we must apply the insights we obtained while researching the questions Q1 and Q2 in the development of countermeasures. Especially interesting in this regard are the limitations of data-only malware (Q1) as well as the lessons learned from existing defenses (Q2) and why they failed or succeed in counteracting data-only malware.

1.3 Contribution

(C1) Detailed Analysis of Data-only Malware. We perform a detailed analysis of data-only malware to identify the capabilities and limitations of this malware form (Q1). To address the question whether data-only malware is an equally powerful and realistic threat as traditional malware, we determine the key properties of traditional malware and employ them as a basis for our analysis. We show that data-only malware is able to achieve *all* of the key properties of traditional malware and surpasses it even in some respects such as its level of stealth. In the process, we also address the problem of computability and illustrate that data-only malware has in many cases the ability to perform arbitrary computations in spite of its dependence on the instructions of the target system. To provide further proof of the practicability of the approach, we additionally present detailed proof of concept (POC) implementations of sophisticated data-only malware capable of infecting current systems. Finally, we identify weaknesses of data-only malware in the form of dependencies, which can provide a basis for the creation of countermeasures against it (Q3).

(C2) Evaluation of Existing Defenses. We conduct an analysis of existing countermeasures and investigate their ability to hinder data-only malware (Q2). We show that hook-based detection is currently the only protection mechanism that can counteract data-only malware.

(C3) Presentation of a Novel Hooking Concept. We analyze the evasion resistance of hook-based detection mechanisms to evaluate their reliability against data-only malware (Q2). In the process, we show that current hook-based detection mechanisms are based on the false assumption that an attacker must modify *persistent* control data to install hooks. This makes these mechanisms vulnerable to evasion attacks. To illustrate this, we present a novel hooking concept that we refer to as *dynamic hooking*. In contrast to existing mechanisms, dynamic hooks exploit vulnerabilities to modify *transient* control data at runtime. The hook itself thereby resides within non-control data. As a result, there is no evident connection between the hook and the control-flow change, which not only permits dynamic hooks to evade existing detection mechanisms, but also makes it generally difficult to create detection mechanisms for them.

(C4) Introduction of a Framework for Malware Detection and Removal. We introduce X-TIER, a framework designed to provide a secure and flexible basis for malware detection and removal (Q3). X-TIER employs virtualization to provide strong security guarantees. Most importantly, our framework enables security applications to remain functional even if an attacker gains full control over the monitored system. To circumvent the semantic gap problem, X-TIER provides the possibility of injecting kernel modules from the hypervisor into a running virtual machine (VM) where they are then securely executed within the untrusted guest system. Due to this approach, security applications implemented on the basis of the X-TIER framework have access to all kernel data structures and functions of the monitored system while they themselves remain strongly isolated from it. By combining security with accessibility, X-TIER can support a wide range of security applications as we demonstrate with various example applications.

(C5) Presentation of Initial Countermeasures against Data-only Malware. We present a systematic approach to counteract the threat of data-only malware (Q3). For this purpose, we introduce a system defense model. The main idea behind this model is to install countermeasures on multiple defense layers within the system so that an attacker must overcome the defenses on *all* layers in order to successfully compromise it. Pursuing this “defense in depth” approach, we propose defense mechanisms on each layer to counteract the threat of data-only malware. To ensure that the proposed mechanisms cannot be easily evaded, we base our novel defense mechanisms on X-TIER (C4) as well as on the dependencies of data-only malware that we identified in the course of its analysis (C1/Q1). We show that this results in an effective defense network which is capable of hindering data-only malware in many cases by presenting the extensive experiments that we conducted to evaluate our approach.

1.4 Target Platform

In this thesis we are primarily focused on the Intel IA-32 and Intel 64 architecture, which are also commonly referred to as x86 and x86-64 respectively. During our discussion we will, for the sake of simplicity, use the term “x86” to refer to *both* the 32-bit and the 64-bit version of this architecture. On the software side, we will primarily consider the Linux and Windows OSs, which are, without doubt, two of the most widely used OSs. Much of what we present can, however, similarly be applied to other architectures and OSs as well.

We will utilize virtualization as a building block for our defense mechanisms. On the hardware side, we will thereby rely on the Intel virtualization extensions and make use of full hardware virtualization [64], while we will use the well-known Linux KVM[2] hypervisor as a software foundation. In the process, we consider the hardware as well as the entire virtualization layer to be part of our trusted computing base. That is, we assume the hardware as well as the hypervisor and the software it requires have been implemented correctly without error and exactly according to their specification. We consider attacks on the trusted computing base out of scope for this thesis. In practice, mechanisms such as HyperCheck [172] and HyperSafe [173] can be utilized to reduce this attack vector.

1.5 Outline

In Chapter 2 we provide the foundation for our discussion of data-only malware as well as its countermeasures. With respect to the former, we cover background information about traditional malware, identify its key properties, and provide an overview of code reuse techniques since they form the basis of data-only malware. For our discussion of countermeasures we provide an introduction to virtualization and virtual machine introspection (VMI) and cover existing malware protection and detection mechanisms.

Based on this information, we will perform a detailed analysis of data-only malware in Chapter 3. To identify the capabilities and limitations of this malware form, we compare it to traditional malware and determine whether it can achieve the same key properties as its predecessor. In addition, we provide sophisticated POC implementations of data-only malware to illustrate the practicability of the approach.

[2]http://www.linux-kvm.org/page/Main_Page

Having obtained a profound understanding of data-only malware, we will, in Chapter 4, consider existing defenses and investigate which of them are effective against data-only malware. In the process, we consider hardware and software based protection mechanisms as well as signature-based detection, anomaly-based detection, integrity checking, and hook-based detection. We show that hook-based detection is currently the only effective countermeasure against data-only malware.

To determine whether current hook-based detection approaches are also reliable, we analyze the existing mechanisms in more detail in Chapter 5. We discuss why current hook-based detection mechanisms are vulnerable and show how the attacker can apply these insights to her advantage to bypass the mechanisms. To this end, we introduce at this point a novel hooking concept, dynamic hooks, that is capable of evading existing hook detection mechanisms.

Having assumed the viewpoint of the attacker, we then take on the role of the defender in Chapter 6 and present our framework X-TIER with the objective of providing a secure and flexible foundation for malware detection and removal. We begin the chapter by stating the goals behind the creation of the framework, before explaining how, based on its design, X-TIER achieves these goals. In addition, we provide example applications which demonstrate that X-TIER is well suited as a basis for a wide range of security applications.

In Chapter 7 we utilize X-TIER and our insights about data-only malware to create countermeasures against this novel threat. For this purpose, we first propose a layered model for system defense. We then discuss defense mechanisms against data-only malware on each layer of the model. To show the effectiveness of the approach, we describe the experiments that we conducted and the detection results that we obtained.

Finally, we draw a conclusion and propose future research directions in the 8th and last chapter of the thesis.

Chapter 2

Foundations

In this chapter, we lay the foundation for our discussion of data-only malware. The key to understand any novel threat is to determine in what ways it differs from the threats that we know. Consequently, to identify the limitations and capabilities of data-only malware, we have to compare and contrast it with *traditional code-based malware*. To follow this approach, however, we require detailed background information about traditional malware first. For this purpose, we begin this chapter with a discussion of traditional malware and its properties.

Besides knowledge about traditional malware, another important prerequisite that we need to fulfill for our analysis of data-only malware is an overview of *code reuse* and the concepts behind it. This is due to the fact that code reuse techniques actually provide the foundation for the implementation of data-only malware. Thus we will follow our discussion of traditional malware by covering the most common code reuse techniques and explaining how they work. The information obtained in the process will, in combination with the background knowledge about traditional malware, serve as a basis for our analysis of data-only malware in Chapter 3.

Having discussed traditional malware and code reuse, we will move on to the topic of malware defense. In addition to providing a comprehensive understanding of data-only malware, a main goal of this thesis is to develop initial countermeasures against this novel threat, should it prove dangerous. Before any countermeasures can be designed, however, we first require a secure and flexible basis that countermeasures can build upon. A technology that seems to be predestined for this task is *virtualization*. To show this we will introduce *VMI* as a key technology for malware detection and removal in the second part of the chapter. We will employ this technology as a basis for designing and implementing a framework for the detection and removal of data-only malware in Chapter 6.

Finally, we will provide a detailed overview of *existing* defense mechanisms against traditional malware. This information will serve as a foundation for Chapter 4, where we will analyze the effectiveness of existing countermeasures against data-only malware.

2.1 Malware

To answer the question whether data-only malware is an equally realistic and powerful threat as traditional malware, we will compare both malware forms in the course of our analysis. For this purpose, we require a solid understanding of existing malware forms and their properties. In this section, we will provide the reader with this information.

2.1.1 Malware Evolution

The Dark Ages (1940s-1970s). The concept of a self-replicating program was first proposed by von Neumann in the 1940s [70, 157] and was later published in his work on "Theory of Self-Reproducing Automata" [171]. The research conducted by von Neumann provides the basis for the first computer virus *Creeper*, which spread at the beginning of the 1970s within the ARPANET, a forerunner of today's internet, and lead to the development of the first anti-virus program, *Reeper*, designed to remove *Creeper* [70]. Around the same time Creeper first appeared, D. J. Edwards [4] identified an attack where operators of a system are presented "with a program so useful that they will use it even though it may not have been produced under their control" [4] leading to the execution of a trap door placed into the program that could then copy user IDs and passwords. Edwards referred to this attack as *Trojan horse*.

The Beginning (1980-1987). While Creeper appeared in the 1970s, the first work in academia that provides concrete examples of self-replicating programs and their properties was presented by Kraus in his diploma thesis "Selbstreproduktion bei Programmen" (self-replication of programs) [77] in 1980. However, at the time the dominant term used for computer viruses remained "self-replicating programs". Two years later, 1982, the first documented outbreak of a computer virus occurred in the form of *Elk Cloner* [70, 108]. The virus was implemented by Richard Skrenta, installed itself into the boot sectors of floppy discs, and targeted the Apple II. In 1983 Cohen then introduced the term *computer virus* [10, 70] that he later in 1984 defined as "a program that can 'infect' other program by modifying them to include a possibly evolved copy of itself" [29]. In the same year, Thompson then extended the work of D. J. Edwards in his well-known article "Reflections on Trusting Trust" [160] and presented the idea of a compiler-based Trojan horse capable of installing a backdoor in programs compiled with it. Two years later, 1986, the *Brain* virus [47] was released by the two Pakistani programmers Basit Farooq Alvi and his brother Amjad. In contrast to Elk Cloner, Brain targeted IBM compatible machines and is thus considered as the first virus for the personal computer (PC) and the disk operating system (DOS) [10]. Later that year, Ralf Burger[1] presented the first file virus for DOS called *Virdem* at a Chaos Computer Club (CCC) meeting in Hamburg that was able to infect COM executables files used by the Microsoft disk operating system (MS-DOS) [70]. In the following year more and more viruses appeared

[1]Burger later also published a book on the subject called "Computer Viruses: A High-tech Disease" [17].

that were now capable of infecting different file types and systems [10]. Since companies were mostly unprepared for this new threat [70], this development also led to the creation of anti-virus software such as *Vaccine*, which was published by Sophos [150] in 1987.

The Rise (1988-1996). On the 2nd December 1988 Robert T. Morris released the first worm on the ARPANET [151]. The so-called *internet worm* infected more than 6000 machines and led to an estimated damage of 96 million U.S. dollars [70]. While the worm itself did not contain any malicious functions, an error in its code led to the reinfection of already infected machines, which finally led to the exhaustion of the infected machines resources and a crash of the system. As a consequence of the incident the U.S. Defense Advanced Research Projects Agency (DARPA) founded the first computer emergency response team (CERT) [43].

In the following years viruses than continued to evolve by using novel techniques and targeting new technology developments. In 1990 the first polymorphic virus *Chameleon* is created, which makes use of encryption to change its appearance. Five years later the first *macro* virus for Microsoft Word appears called *Concept*. In 1996 the first virus for Windows 95 is observed and one year later viruses for Linux and Office 97 appear.

The Outbreak (1996-2005). During the course of the next three years, this trend continues and viruses, Trojans, and worms start to target more and more technologies including Windows 98, Office, Outlook, HTML, Java, and E-Mail [70]. In addition, Trojan horses such as *Back Orifice* surface, which provide their users with full control of the infected machines. With the widespread use of the internet, the creation of classical viruses then diminishes and malware authors start to focus on network-based viruses and worms. This leads to large scale infections through viruses such as "*ILOVEYOU*" [126] or the *Code Red* [96] and the *Blaster* [6] worms. While affecting a large number of machines, the malware appearing at the time still had no primary malicious intent though. Instead, the malware could be mostly considered as a prank.

The Business (2005-Present). This changed in the second half of the 2000s when malware was no longer viewed as a technical gimmick, but rather as a tool for cyber espionage, a weapon for cyber warfare, and a business model for cyber criminals. Due to the latter, a new malware form referred to as *bots* appeared, which would not only spread, but also provide its creator with full control over the infected machine turning it into a *zombie*. In addition, malware started to hide itself within the system to avoid detection. This trend led to a second major new malware form referred to as *Rootkits* [13, 58]. In the following years, malware started to evolve faster and faster making it more and more difficult for anti-virus vendors to keep up. Malware in virtually any language appeared aimed at a wide variety of platforms ranging from embedded devices to server systems.

The Future (2009-Present). In 2009 Hund, Holz, and Freiling [61, 62] presented the first *data-only* malware in form of a return-oriented rootkit. This work was extended by Chen et al. [24] in 2010, who demonstrated a jump-oriented data-only rootkit. Besides our own work, however, this remains the only work on data-only malware to date. We are the first to study the properties, capabilities, and limitations of data-only malware as well as its effects on current protections mechanisms. While data-only malware has not yet been spotted in the wild, the sophistication level of recent attacks such as Stuxnet, Duqu, and Flame [9] suggest that this malware form is just beyond the horizon.

2.1.2 Types

In practice the term *virus* is often used as a synonym for malware. In fact, however, a virus represents a specific type of malware. In the following we will introduce and differentiate five of the most important malware classes. Namely, viruses, worms, Trojan horses, rootkits, and bots. We will discuss the relationship between these classes and the reasoning behind their selection in the next section of this thesis.

Virus. The *computer virus* is one of the oldest[2] types of malware, but is in its classical form only seldom encountered nowadays [44, 70]. Similar to its medical eponym, a computer virus is an infectious agent whose primary purpose is to *replicate* [13]. In contrast to medical viruses which infect living cells, however, a computer virus replicates by infecting other programs [29]. To do so the virus attaches itself to a program and manipulates it in such a way that it is executed whenever the *host* program is launched. In the simplest form, this can either be achieved by modifying the entry point of the infected program such that it points to the virus instead of the host [43] or by placing the virus at the beginning of the infected file [70]. More advanced mechanisms manipulate the data structures or the instructions of the host to get executed [157]. Besides infecting executables [10], computer viruses can also infect boot sectors (*boot virus*) and data files (*macro or data virus*) such as Microsoft Word documents or pictures by making use of macros or vulnerabilities [43].

An essential property of a virus, which can also be used to distinguish it from other malware forms, is that it does not replicate on its own [43, 44]. Instead the replication process must always be triggered by an external entity e.g. by a user that executes an infected program [70]. Once triggered, however, the replication itself must then be performed by the virus alone without relying on any external help. The replicated form of the virus may thereby differ from the original. That is, a computer virus may *mutate* during replication, which can significantly raise the bar for its detection.

Worm. Similar to a virus, a *worm* is an infectious agent whose main purpose is to replicate itself. In contrast to a virus, however, a worm is *self-propagating* [13, 44, 158]. Consequently, it does *not* require an external entity that initiates the replication process, but rather propagates on its own. This property provides worms with the capability to infect a large number of systems in a short period of time. For instance, have researchers shown that so-called *flash* worms could infect “almost all vulnerable servers on the internet in less than thirty seconds” [154], if they apply advanced strategies

[2]The first research in the field of computer viruses was actually conducted by John von Neumann in the 1940s [70, 157] and was later published in his work on “Theory of self-reproducing automata” [171].

for identifying vulnerable hosts. An example of such a strategy could be a "hit-list" of vulnerable hosts that is created in advance before the worm is launched and enables the worm to quickly infect a critical mass of hosts, which can then in turn be used to infect other systems.

Due to their similarity, it is not always straight forward to distinguish between a virus and a worm in practice. Since worms spread on their own, they in general do not require a host to function [43]. However, as there are worms that also infect files similar to viruses [157] (e.g. to be able to infect systems that are not connected to the same network), this attribute alone is not sufficient to distinguish a worm from a virus. Other authors [10, 157] distinguish between a worm and a virus based on the fact that worms spread over the network, while viruses usually only infect local files. This categorization based on the propagation method, however, falls short in the case that a virus spreads over a network drive [70].

In this thesis, we will use the *infection mechanism* to differentiate between a worm and a virus. While a worm is able to automatically infect other machines using vulnerabilities on the target system, a virus can only replicate if it is activated by an external entity.

Trojan Horse. According to Bishop [10], a *Trojan horse*, sometimes simply referred to as a Trojan, is "a program with an overt (documented or known) effect and a covert (undocumented or unexpected) effect" [10]. Consequently, a Trojan horse consists – similar to the historical original – of two elementary parts: A visible or known part (the wooden horse) and a hidden part (the soldiers in it). Since the user does not expect the hidden part, she can be tricked into executing the seemingly benign program, which will then in turn activate the hidden malicious functionality. This implies that a Trojan horse like a virus requires an external entity to activate it (the Trojans that pull the horse into the city). In opposition to a virus, however, a Trojan horse does not replicate.

Rootkit. In contrast to other malware forms a *rootkit* is not used to infect a target system, but is rather installed once an attacker already controls the target system. The goal is thereby to provide sustained access to an infected machine [167]. To achieve this the rootkit tries to hide all signs of an infection such that the system seems normal to an observer, while it is in reality controlled by an attacker. That is, the sole purpose of a rootkit is to provide *stealth*.

In the simplest form, rootkits are installed by replacing important system utilities such as `ps` or `ls` with attacker controlled binaries. In contrast to the original binaries, the replacements will seem to operate normally, but will hide compromising information from the user. Each replaced program can thereby be considered as a Trojan horse that provides the standard functionality, but contains additional hidden functionality. For instance, may the attacker controlled version of `ps` not display processes with a specific PID or name. Since such rootkits solely operate in user space, the are also referred to as *user space rootkits*.

More sophisticated and a lot more dangerous are so-called *kernel space rootkits*, which directly infect the kernel of the system. As a result, this type of rootkit runs at the same privilege level as the OS allowing it to modify any part of system including the kernel itself. This enables the rootkit to hide compromising processes and files by directly modifying kernel data structures [152]. Due to the size and complexity of modern kernels detecting such rootkits is a very challenging task.

Bot. Originally *bots* were not malicious in nature, but rather helper programs used within chat networks such as the internet relay chat (IRC) [30]. The main purpose of a bot in this context was to support channel administrators or to provide information and entertainment for users. Since bots were in general able to perform predefined actions on their own and could, for instance, react to chat messages, the term bot eventually established as a short form of *robot* [59]. This capability was later then abused to create *attacking* bots that in contrast to their predecessors were programmed to attack users, channels, or servers [30]. At the time these bots were, however, in general isolated and operated on their own. With the rise of the internet and the discovery of malware as a business model, bots then left their original battlefield and evolved into *worm-like* programs that could infect other hosts on the network. Once infected, the bot would provide its master with full control over the host turning it into a *zombie* completely at the mercy of its master. As the bot continues to spread and infects other machines, this will provide its owner eventually with an army of zombie machines forming a so-called *botnet*. Since a botnet can potentially consist of thousands of machines [59], they enable their owners to launch powerful large scale attacks such as DDOS. In addition, bots often rely on similar stealth techniques as rootkits to avoid detection and to maintain access to infected machines.

2.1.3 Type Relationship

Surprisingly, defining a relationship between individual malware types is not as straightforward as one might think. On the one hand, this is due to the fact that there is no clear terminology for each individual malware type in literature. Quite the contrary, the definitions for each malware class may vary significantly from author to author. Bishop, for instance, considers a computer virus to be a type of a Trojan horse [10], while other authors such as Szor [157] clearly distinguish between the former and the latter.

On the other hand, the lines between the individual malware types get more and more blurry [68]. Nowadays most malware that is encountered is usually not a pure representative of a specific malware class, but rather a combination of different malware types. A *bot*, for example, can often spread on its own like a worm, but at the same time it uses stealth techniques formerly only found in rootkits to avoid detection. Clearly, this malware type is therefore neither a worm nor a rootkit, but rather a combination of both species. This raises the question whether the approach of dividing malware into individual types is timely.

While *classifying* modern malware into types may be difficult, working with malware types instead of concrete instances has an important advantage with regard to the research problems we consider within this thesis: one of the key questions that we try to answer is whether data-only malware is as powerful as traditional malware and in which regards it is inferior or superior to its traditional counterpart. To address this question, we must *compare* both malware forms. However, which malware instances should we select for this comparison? Clearly, we cannot include each and every malware sample into our considerations. Instead, we must select a subset of traditional malware instances that consolidate the *core properties* of this malware form such that the results of our comparison are not only valid for a few malware instances, but are meaningful for traditional malware as a whole. Making use of malware types instead of malware instances provides an elegant solution to this problem. As described in the beginning, each complex malware instance can essentially be seen as a combination of classical malware types. The complex instance thereby *inherits* the properties and limitations of its *base types*. Consequently, by working with base types instead of individual instances, we can include the core properties of a wide range of malware samples in our analysis while we at the same time only have to consider a few types.

Following this idea, we now introduce a plain relationship model based on the malware types defined in the last section. The main idea behind this model is to divide today's complex malware forms into base types. The quality of our model thereby heavily depends on the base types we select. Since the model should permit conclusions for malware in general, selecting base types that have a significant impact on the properties and capabilities of other malware forms is crucial. While the definitions of individual malware types vary, many authors (e.g. [10, 43, 70]) distinguish between three fundamental malware types: viruses, worms, and Trojan horses. Each of these malware types provides essential properties that are often found in today's malware:

Viruses	Replicate and infect other *files*, but require an *external entity* (host) to spread.
Worms	Replicate on their *own* using *vulnerabilities*.
Trojan Horse	Introduce *hidden* functionality into benign *programs*.

However, a relationship model defined on these three malware types alone would be incomplete. Besides infection strategy and hidden functionality another important aspect of today's malware is *stealth* [13, 58]. Since most widespread malware nowadays has a financial or military motivation, the goal of the malware is not just to infect a machine, but to do so unnoticed. Therefore *rootkits*, whose sole purpose is to provide stealth, must be considered as another important malware base type.

Using these four malware types, we can define the following type relationship model: most malware instances are derived from one or a combination of the four malware base types virus, worm, Trojan horse, and rootkit. Consequently, to compare and contrast traditional malware from data-only malware, it is sufficient to analyze the properties, capabilities, and limitations of these four base types. How well the proposed model can be applied to modern malware types can be amply demonstrated if we reconsider the example of a *bot*, which is one of the most widespread malware types today [159]. A *bot* combines autonomous infection with remote control functionality and stealth. Consequently, bots "can be seen as a combination of worms, rootkits and Trojan horses" [59], which can be expressed by the proposed model. The interested reader can find a more thorough discussion of malware type relationship models in [69].

2.1.4 Key Properties

In the last section we introduced a type relationship model for traditional malware. In the process, we determined that the four malware base types virus, worm, Trojan horse, and rootkit exhibit many of the key properties of traditional malware. So far, however, we have not discussed these properties. In this section, we will introduce the properties that are most relevant for our discussion of data-only malware.

2.1.4.1 Infection Mechanism

One important aspect of malware is the infection mechanism that it can leverage [157]. That is, how does malware actually get onto a system. In the case of traditional malware, we can distinguish between two basic infection mechanisms: On the one hand, malware may be able to infect a victim through the exploitation of some *vulnerability*. To use this attack vector, the target system must provide a control-flow modifying vulnerability that can be exploited by the malware to load and execute itself and the vulnerable application must be running on the target system at the time of the infection. The exploitation of the vulnerability can thereby occur with (e.g. rootkits) or without (e.g. worms) human involvement.

On the other hand, the attacker can send the victim a specially crafted file and deceive the victim into executing it (e. g., malware that spreads as email attachments). Such a *file-based* infection approach is, for example, commonly used by a virus [157]. How this infection mechanism can be realized thereby heavily depends on the format of the file (e.g. Executable and Linkable Format (ELF)) that the malware infects.

Finally, note that we consider a file that is launched by a user and triggers a vulnerability on the victim's system to be a vulnerability-based infection mechanisms and not as a file-based infection mechanism. A file-based infection mechanism does *not* require a vulnerability for the loading and the execution of the malware.

2.1.4.2 In-Memory Strategy

Besides the infection mechanism, another crucial aspect of malware are the different strategies it uses for its execution. Particularly interesting in this regard is the *in-memory* *strategy* that malware forms leverage. Depending on whether the malware remains in memory after initialization or if it removes itself directly after its initial execution, one can thereby once more distinguish between two types of malware: *direct-action* malware or (memory) *persistent* malware [157]. In the following we will discuss both types and their properties in more detail.

Direct-Action Malware. In contrast to persistent malware, direct-action malware does not permanently load itself into memory. Instead the malware is just executed once, performs the desired action, and exits. Such attacks are executed by file-based viruses, for instance. Once the infected file is executed, the virus looks for another file that it can infect, executes its infection routine (given that it could identify a target), and exits.

While the effects of direct-action malware might last (e. g., the modification of a data structure), there is no way for the malware to respond to further actions within the system. That is, direct-action malware is incapable of intercepting *events* that occur within the infected host and could, for instance, not be used to implement a key logger.

Although this effectively limits the usage scenarios for direct-action malware, this malware type also has an important advantage. Due to the fact that this malware stores no permanent structures within memory the detection of direct-action malware can be difficult especially if the malware does not store any permanent data on disc either. In this case, direct-action malware can only be detected during the short period of time of its execution.

Persistent Malware. Malware in general needs to intercept events within the system to be able to fulfill its purpose [116, 175]. Consider rootkits for example. Without the capability of event interception, this malware type would be severely limited and could not even provide basic functionality such as key logging and file hiding. Event interception, however, requires malware to run *persistently* within the memory of the infected system. Due to this fact, persistence is one of the most important properties of malware today.

For the purpose of this thesis, we consider malware to be *persistent* when it makes permanent changes in memory and permanently changes the *control flow* within a system such that it can continue to achieve its objective. This characteristic allows the malware to be aware of and react to changes in the system. The simplest example of such functionality is replacing a function pointer with a pointer to a malicious function that collects the data being input to the original function. We will discuss the various possibilities that exist for malware to permanently divert the control flow in more detail in Section 2.4.2.5.

2.1.4.3 Resident Malware

The astute reader will have noticed that the above provided definition for *persistence* does not include any changes to non-volatile data storages. As a result, *persistent* malware may, according to its definition, be unable to survive reboots. The reason for this apparent lack in the definition is that we make a clear distinction between *persistent* malware and *resident* malware. More precisely, we consider any malware that has the ability to continue to achieve its objective without any human interaction despite a reboot or power cycle as resident in the system and therefore as *resident* malware. It goes without saying that residence is similar to persistence an essential property of malware.

2.1.4.4 Stealth Taxonomy

Having discussed fundamental properties of malware such as the infection mechanism and the in-memory strategy, we will now move on to consider more advanced properties of malware that may not be crucial for the execution of the malware, but are often essential for it to be able to achieve its goals. The first property that we want to consider in this context is *stealth*.

Since a primary goal of malware is to avoid detection, stealth is a crucial capability that many malware forms strive for (e.g. rootkits). To measure the level of stealth that a malware achieves, we will leverage a stealth malware taxonomy that was first proposed by Rutkowska [131]. The fundamental idea behind this taxonomy is to define the level of stealth of a malware form based on the changes it conducts to the system. As a simple example consider malware that only changes data within the system and malware that modifies code within the system. Clearly, the former malware type provides more stealth than the later, since changes to code areas, which are usually static, can much easier be detected than changes to data areas which may constantly change.

Rutkowska distinguishes between four different malware types. In the following, we will describe each of these malware types in more detail.

Type 0 Malware. Type 0 malware is the simplest and least stealthy malware type in the taxonomy. It is shown in Figure 2.1. In contrast to other malware types, type 0 malware does *not* modify other processes or the OS. Instead, the malware type runs within a process of its own. An example of such a malware type could be a Trojan horse that is launched by a user.

To detect this malware type we need to decide whether a given binary is malicious or benign. While this problem is in general undecidable [29], it can be solved in many cases in practice. Antivirus software, for instance, makes use of various mechanisms such as signatures[3] to detect malicious binaries.

Type 1 Malware. Type 1 malware changes system resources that are usually *constant* such as code areas. For instance, could a rootkit modify the code region of a process to install a hook[4] that will trigger its execution. This approach is shown in Figure 2.2.

[3]We will cover signature-based detection mechanisms in more detail in Section 2.4.2.2.

[4]Hooks will be discussed in more detail in Section 2.4.2.5.

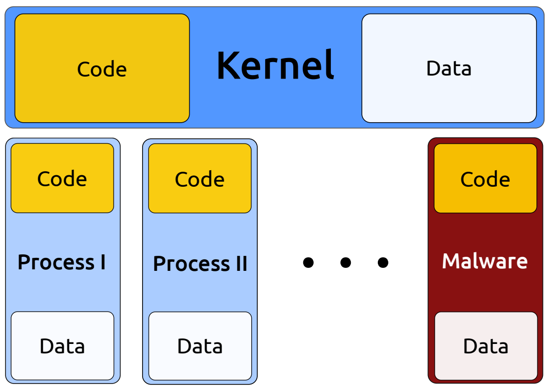


Figure 2.1: Type 0 malware does not change any code or data sections but runs as an own process within the system.

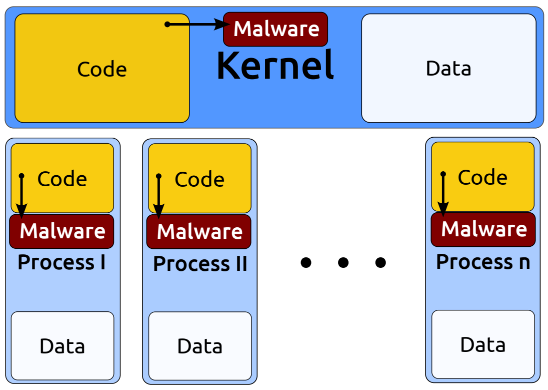


Figure 2.2: Type 1 malware changes constant system resources such as code sections.

Since type 1 malware could constraint itself to solely modify system resources in memory, it is insufficient to scan binaries on disc to detect it. Instead, the memory of the system must be searched as well. In general, distinguishing between benign and malicious modifications to system resources is a difficult task though. Since type 1 malware changes constant resources, however, the problem can in this case be significantly reduced: to detect malicious changes, it is sufficient to determine whether a constant region has been modified. A possible approach to achieve this is to calculate a cryptographic checksum of all constant system resources. Changes can then be detected by periodically recalculating the checksum and comparing it to its original value. If the values differ, the resource was changed.

Type 2 Malware. Similar to type 1 malware, type 2 malware changes resources within the system to provide its malicious functionality. The main difference between the two malware types is that type 2 malware *only* changes *dynamic* system resources. Consequently, simply detecting a change is no longer sufficient to detect the malware type. Instead, we must find a way to verify the integrity of resources that may constantly change during normal operation such as data areas. Naturally, this is a much more difficult problem, which is why type 2 malware is significantly harder to detect than type 1 malware. Rutkowska even goes as far as to say that the detection of this malware type is currently in general impossible, since existing 'systems are not designed to be 'verifiable'" [131]. An overview of type 2 malware is given in Figure 2.3.

Type 3 Malware. The last and arguable most stealthy malware type in the taxonomy is type 3 malware. Similar to type 0 malware this malware type does not modify any system resources, but is still able to control the entire system. To achieve this, the

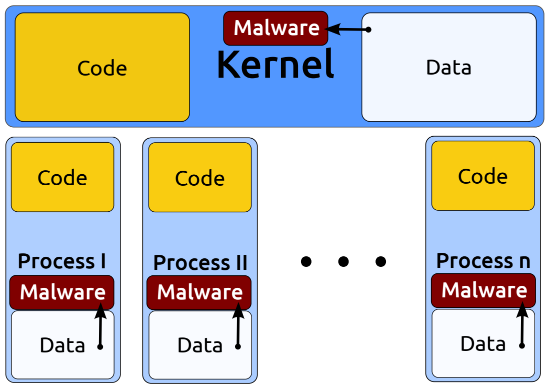


Figure 2.3: Type 2 malware. In contrast to type 1 malware, type 2 malware only changes dynamic system resources.

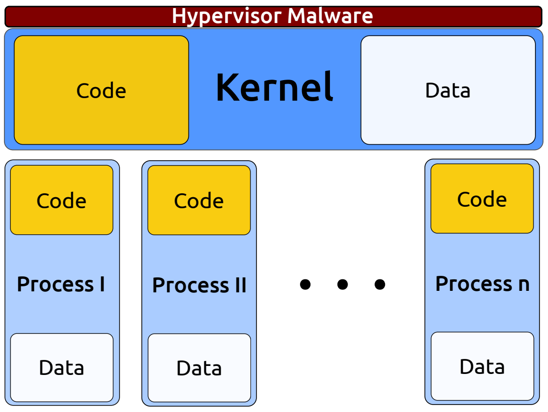


Figure 2.4: Type 3 malware. To avoid modifying system resources, type 3 malware places the entire target system into a virtual machine.

malware type makes use of virtualization[5]. By placing the victim's system into a VM, the malware can operate from the hypervisor level where it has access to the entire state of the system and can perform arbitrary actions without the target system being aware of its presence. This approach is shown in Figure 2.4.

To detect type 3 malware, one must detect the presence of a hypervisor from within a guest system. While this could, for instance, be accomplished with the help of timing attacks providing reliable detection mechanisms poses a difficult problem, since malware authors could implement countermeasures on the hypervisor level that mitigate or disable detection mechanisms running within the VM. After all, in this scenario the malware runs at a higher privilege level than the guest OS making virtually any detection mechanism vulnerable to attacks.

2.1.4.5 Environment Dependencies

In general, malware is unable to infect arbitrary systems, but rather requires a specific software and/or hardware configuration to execute [157]. That is, malware *depends* on a particular *environment* to run. This dependency heavily influences the characteristics of malware. For example, malware written for the x86 architecture will not be able to infect ARM-based devices[6], since the instruction sets of both architectures significantly differ from each other.

Environment dependencies are especially interesting for our analysis as they reflect the limitations of a particular malware form. As a consequence, they often provide a

[5]We will discuss virtualization in more detail in the Section 2.3.

[6]Unless the malware was especially designed for both architectures and is able to generate an ARM version of itself of course.

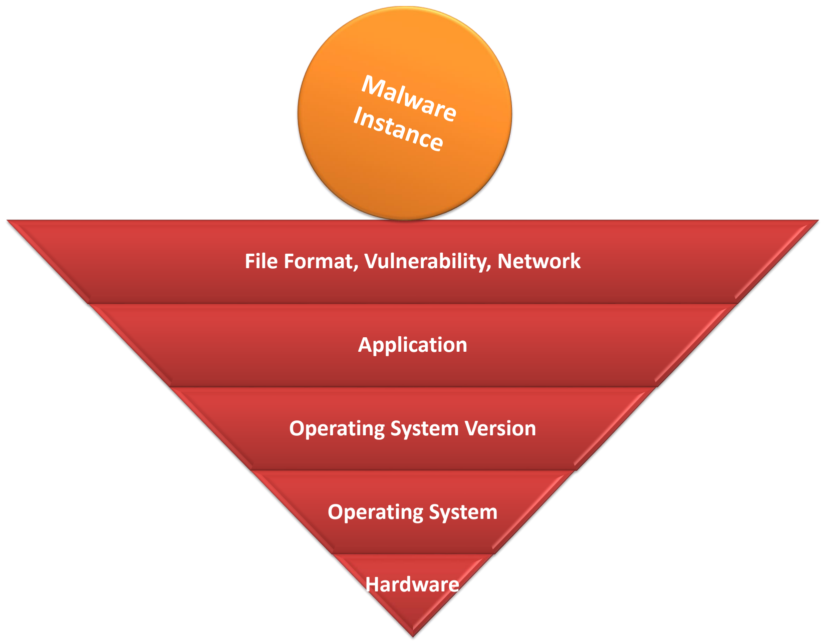


Figure 2.5: Important environment dependencies that a malware may have. The lower a dependency's position within the pyramid, the higher is its complexity and consequently the more difficult is it to resolve.

basis for countermeasures. For instance, if a malware instance requires the existence of a particular vulnerability to infect a system, we can stop the malware from spreading by fixing the vulnerability. In the course of our analysis, we should thus particularly focus on identifying the dependencies of data-only malware, as they could prove invaluable in defending against this novel threat. For this purpose, we will in this section provide an overview of important environment dependencies that malware may have. We will then analyze these dependencies in the context of data-only malware in Section 3.6.3.

The environment dependencies that we want to consider are shown in Figure 2.5. In the following, we will briefly describe each of them. In the process, we will move from the most complex dependency (Hardware) shown at the bottom of the upside-down pyramid to the least complex dependencies (File Format, Vulnerability, Network) shown at the top. The less complex a dependency, the easier is it to resolve. The interested reader can find an even more exhaustive list of environment dependencies in [157].

Hardware Architecture Many malware forms require a particular hardware architecture to execute. For example, they generally depend on a particular *instruction set* and a specific central processing unit (CPU).
To resolve this dependency malware must either be able to generate an architecture compatible version of itself

(for each supported architecture) or it must make use of emulation to simulate a particular architecture. In the latter case, however, at least the emulating component must exist for each of the target architectures.

Operating System

Besides the hardware architecture, most malware is designed for a particular OS. This is due to the fact that each OS exports a specific application binary interface (ABI)[7] that enables user space programs to communicate with the OS via system calls. Since, similar to normal programs, user space malware needs to make use of this interface to access the hardware, it generally depends on a specific ABI. This is even truer for malware that resides within kernel space and makes use of the internal functions that the OS provides.
A possible way to resolve this dependency would be if the malware was to infect the kernel and would then provide its own drivers to communicate with the hardware.

Operating System Version

Especially kernel space malware may not only depend on a particular OS, but may even depend on a particular version of an OS. The reason for this is that internal functions of the OS may change without notice between different versions, since the functions are only used by the kernel and are inaccessible from user space. Consequently, if malware relies on a particular internal function, it may be unable to execute if the function is updated in a new OS version.
To reduce this problem, malware could only rely on exported functions that cannot be easily changed between different OS releases.

Application

Certain malware may require particular applications to execute. Consider, for instance, a malware written in an interpreted language such as Python. If the malware is not natively compiled for a specific architecture, the malware is only able to run if the required interpreter is present on the target system.
Malware could overcome this problem by shipping the application that the malware requires with the malicious binary.

[7]We will cover the ABI in more detail in Section 2.3.1.1.

File Format	A classical virus often depends on specific file formats for replication. For instance, a virus may only be able to infect Portable Executable (PE) [93] or ELF [155] executables. To resolve this dependency a virus must provide infection and startup code for each executable it wants to infect.
Vulnerability	Self-propagating malware may require a specific vulnerability on the target system to spread. For example, a worm that uses a newly discovered vulnerability to infect a large number of hosts on the Internet. To reduce the dependency malware can make use of multiple infection mechanisms such as vulnerabilities, files, and brute forcing of accounts.
Network	Finally, certain malware may only able to execute if the target system is connected to a network. For instance, a bot may be unable to receive commands if the infected system is not connected to the Internet. In this case, the bot may only be able to fulfill a subset of the intended functionality. To resolve this dependency malware must find a way to establish a connection even if the infected machine is not directly connected to a network. This can for example be achieved with radio signals [55].

2.1.4.6 Encryption, Polymorphism and Metamorphism

The last property that we want to consider is the capability of malware to *evade* detection. A detection approach that is particularly interesting in this regard is *signature-based* detection as it is the approach that is most commonly leveraged by antivirus software. The basic idea behind signature-based detection is to compute a signature for each known malware instance. Once created, the signature can then be used to identify malicious executables by checking whether the signature matches on a given executable. The signature thereby essential represents an unique identification pattern[8]. Modern malware, however, leverages various techniques to evade the signature that has been computed for it. In the following we will provide an overview of these techniques.

A possible way to evade signatures is to make use of *encryption* [10, 99, 157, 180]. By encrypting the malware with different keys, each of the resulting ciphertexts will look different. Consequently, it becomes quite difficult to generate an unique signature for the malware. However, for this approach to work, the malware requires a decryption routine

[8]We provide a more detailed description of signature-based detection in Section 2.4.2.2.

that can decipher the encrypted malware body at runtime. While the malware body can be encrypted, the deciphering routine must remain unencrypted. Thus instead of computing a signature for the encrypted malware body, it is possible to create a signature for the deciphering component. To solve this problem malware authors came up with the idea of *mutating* the deciphering routine. This approach is known as *polymorphic* malware [10, 99, 157, 180].

Polymorphic malware makes use of a mutation engine that allows it to change its form. The task of the mutation engine is thereby to change the code of the malware while keeping its original functionality intact. The most common techniques used for this purpose are *junk code insertion*, *instruction replacement*, *instruction permutation*, *variable/register substitution*, and *code transposition.* [14, 75, 125]. In the following we will describe each of these mechanisms in more detail.

Junk Code Insertion. The main idea behind junk code insertion is to alter the appearance of code by inserting additional instructions that do not alter the logic of the program. For instance, could a malware author first add one to a specific register and then immediately subtract one from the same register with the next instruction. Clearly, the value within the register would be the same after both instructions have been executed, but the added instructions may break an existing signature for the malware.

Instruction Replacement. Another possibility to change the structure of code without affecting its logic, is to *replace* existing instructions with different but equivalent instructions. As an example assume that the malware contains the following x86-64 assembler instruction:

```
mov rax, $0x0
```

The purpose of this instruction is to move the value "0" into the register `rax`. Clearly, this is not the only instruction that is capable of achieving this goal. Instead, a mutation engine could, for example, substitute this instruction through one of the following instruction blocks:

```
; Frist Possibility
and rax, $0x0

; Second Possibility
xor rax, rax

; Third Possibility
or rax, $0xffffffffffffffff
add rax, $0x1
```

In all cases the register `rax` will contain the value zero at the end, however, the instructions that are used to achieve this are different each time. It goes without saying that there exist tons of other possibilities that could be used to substitute the original instruction stated above. The given instruction blocks are merely an example.

Instruction Permutation. Instead of replacing instructions, it is also possible to reorder instructions to change the layout of the code. To use this approach, the mutation engine can, for instance, select instruction pairs that are independent from each other and the following instructions and can thus be swapped. Consider the following example:

```
1    mov rax, $0x42
2    mov rbx, $0x1337
3    add rax, rbx
```

In this case, the instruction in Line 1 can be exchanged with the instruction in Line 2 without altering the result of the computation.

Variable/Register Substitution. Besides replacing entire instructions the mutation engine can also alter the registers that instructions operate on. Similarly, the location were specific variables are stored in memory can be changed by altering the layout of the data section of the malware and updating the instructions in the code accordingly.

Code Transposition. Similarly to permuting individual instructions, it is also possible to reorder entire code blocks or to split code blocks in additional blocks. This is usually achieved by inserting additional branch instructions that will ensure that the original control flow remains intact, while the appearance of the code changes drastically. The following pseudo code example demonstrates this approach:

Listing 2.4: Code transposition (before)

```
start:
  Instruction 1
  Instruction 2
  Instruction 3
  Instruction 4
  Instruction 5
```

Listing 2.5: Code transposition (after)

```
3:
  Instruction 3
  JUMP 4
2:
  Instruction 2
  JUMP 3
start:
  JUMP 1
5:
  Instruction 5
1:
  Instruction 1
  JUMP 2
4:
  Instruction 4
  JUMP 5
```

All of the above described mechanisms cannot only be applied to the binary level, but can also be performed on a higher level such as the source code, for example. In addition, it is also possible to substitute entire algorithms/functions within the code as long as they produce the same result. To alter the entire decryptor, malware could, for instance, make use of different encryption algorithms. However, while these techniques allow the malware to create a wide range of different decryptors, the problem that remains for polymorphic malware is that the malware body itself actually remains unchanged. From this it follows that a signature for the malware body will still match, once a polymorphic malware has been decrypted, which has to happen at some point in time during the execution of the malware. This is where *metamorphic* malware comes into play.

In contrast to polymorphic malware, metamorphic malware does not just mutate the decryptor of the malware, but the entire malware including the mutation engine. Thus a metamorphic malware can essentially be seen as a body-polymorphic malware [157]. Metamorphic malware in general always alters its appearance when it replicates. Consequently, two copies of the same malware never look the same even during execution.

2.1.5 Summary

We provided a type relationship model for traditional malware that states that each complex malware instance is derived from one or combination of four base types. Namely these base types are Trojan horse, virus, worm, and rootkit. Based on these base types we described the properties of traditional malware that are most interesting for our analysis. These properties can thereby be divided into two groups: *fundamental* properties that are crucial for the actual functionality of malware and *advanced* properties that while not essential for its functionality are often vital for its objectives. Infection mechanism, in-memory strategy, and residence belong to the former category. Its level of stealth, its environment dependencies, and its ability to evade signature-based detection belong to the latter.

2.2 Code Reuse

The fundamental idea behind data-only malware is to control the execution flow of a system solely based on data. For this purpose the attacker introduces an especially crafted data structure into the system that *combines* existing instruction sequences into a new program and *controls* its execution. That is, the malware effectively assembles its own program code by *reusing* the instructions that already existed before its presence. *Code reuse techniques* thus provide the foundation for the creation of data-only malware. In the following, we will provide an overview of existing code reuse techniques and explain how they work.

2.2.1 Techniques

Code reuse techniques were originally developed as exploitation primitives to circumvent code-based protection mechanisms. When exploiting an application the general goal of an attacker is to execute arbitrary code. Before the existence of code-based protection mechanisms, achieving this task was often straightforward as the attacker could simply place her shellcode[9] in an arbitrary *data* area and execute it from there. In his famous article "Smashing The Stack For Fun And Profit" [3] Aleph One, for instance, places the shellcode directly onto the stack, which was marked as writable and executable at that time. With the introduction of protection mechanisms such as $W \oplus X$ (see Section 2.4.1.2 for details) this attack vector was mitigated, however, and data areas were no longer marked as executable. As a result, attackers were forced to find a different way to execute arbitrary code. This development led to the introduction of the first code reuse technique in 1997 that its creator Solar Designer coined "ret2libc" (return to libc) [149].

2.2.1.1 Return to Libc (ret2libc)

The main idea behind ret2libc is to make use of the functions that are already present within the memory of a vulnerable application instead of introducing own code. To understand the technique, one must keep in mind that a function is essentially a reusable piece of code that performs a deterministic action based on its arguments. By invoking a function with attacker controlled arguments, the attacker can thus use it to perform an action on her behalf without needing to introduce code.

To use ret2libc, an attacker must first place the arguments of the function she wants to invoke in the foreseen register/memory locations as specified by the ABI. In the case of a 32-bit Linux system running on the x86 architecture, for instance, the arguments would thereby be stored on the stack in reverse order. Next, the attacker must set the instruction pointer (IP) to the virtual address of the function that she wants to invoke. This can, for example, be achieved by overwriting a return address on the stack. As soon as the overwritten return address is used, the function call will occur and the desired operation will be performed. To spawn a shell, the attacker could, for instance, make use of `system('/bin/sh')` instead of writing her own shellcode.

A question that immediately comes to mind though is whether the functions that a vulnerable application provides are sufficient for the actions that an attacker requires. What is important to understand in this regard is that an attacker is generally not restricted to the functions of the vulnerable application alone. Instead, the attacker is able to use *any* function residing within the memory of the vulnerable application including *library functions*. Since almost every application requires external libraries to run, this provides the attacker with a much lager attacker surface. In case of Linux, for instance, the attacker can make use of the entire GNU C library (libc) to accomplish the desired task, which is why the technique is referred to as return to libc.

Due to the numerous functions contained within libraries today, ret2libc is a very powerful technique. For instance, can the attacker allocate a new memory region that she can mark as *writable* and *executable*, which then allows her to execute arbitrary code on a system in spite of $W \oplus X$. This makes the technique predestined for the

[9]Shellcode is a general term that is used to refer to the instructions that are executed by an attacker once she gains control over the system during the exploitation of a vulnerability. Since the attacker usually spawns a shell in this situation, the term shellcode established.

first step during an attack. In addition, the attacker is not limited to a single function call, but is able to chain an arbitrary number of function calls together [100]. This is achieved by creating a *control structure* containing multiple function frames on the stack. Each frame will thereby contain the address of the function to invoke as well as the arguments to the function. By careful manipulation of the frame pointer (FP) each return of an invoked function will load the next frame and trigger the execution of the next function. This enables an attacker to perform sophisticated computations. In fact, given a large enough codebase, ret2libc has even been proven to be capable of achieving Turing completeness[10] [162]. In addition, ret2libc is hard to detect in practice and is considered as a general limitation of control flow integrity validation (CFI) mechanisms [186], since it is in general difficult to distinguish regular function calls from function calls simulated by an attacker.

2.2.1.2 Return-Oriented Programming (ROP)

While libraries provide a wide range of functions, they certainly do not contain a dedicated function for every action that an attacker wants to perform. For example, it is unlikely that a function exists that hides a process, since this functionality is obviously not needed by a benign program. While the desired action can be achieved by combining multiple function calls, this may be cumbersome and require a lot of space for the individual function frames on the stack. This is why attackers came up with a second code reuse technique referred to as *return-oriented programming (ROP)* [138].

The fundamental idea behind ROP is to create a new program by combining small instruction sequences instead of entire function as in the case of ret2libc. The execution order of these instruction sequences is once more controlled by a control data structure. Naturally, not every instruction sequence is suited for use with ROP. To be able to use an instruction sequence as a building block it must end with a `ret` instruction. Such instruction sequences are referred to as *gadgets*. The property of this instruction that makes it so useful is that it pops the top value off the stack into the IP redirecting the control flow to the address that was on the top of the stack. By carefully constructing the stack, it is therefore possible to execute sequences of gadgets one after another as shown in Figure 2.6. This is achieved by placing the memory addresses of the gadgets in the order on the stack in which they should be executed. Since every gadget ends with a `ret` instruction, the final instruction of each gadget starts the execution of the next gadget by getting its address from the stack and placing it into the IP.

To make this scheme work, the stack pointer (SP) must initially point to a control structure containing the addresses of the individual gadgets, often called a *ROP chain*. In the simplest case, this can be achieved by directly copying the ROP chain in the stack such that the saved IP is overwritten by the address of the first gadget. However, this may not always be possible as the available space in the stack may be limited. In such a

[10]We will cover Turing completeness and code reuse in more detail in Section 2.2.2 and Section 2.2.4.

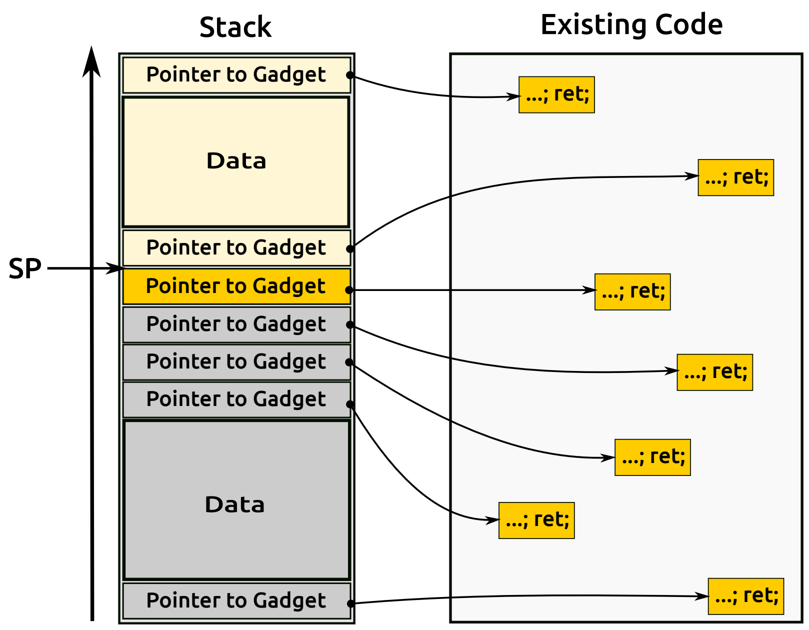


Figure 2.6: The layout of a ROP control structure. The stack pointer (SP) functions as program counter. The gadgets that have already been executed are shown in grey, while gadgets that will be executed in the future are shown in light yellow. Each gadget ends with a `ret` instruction. The current gadget is shown in yellow.

case, one must copy the ROP chain somewhere else in memory and point the SP to this location. This is achieved through a *stack pivot* sequence.

A stack pivot sequence generally only consists of a few gadgets or, in the worst case, only one. Setting the SP to the beginning of the ROP chain in such a scenario is a very challenging task. Achieving this is often only possible if the attacker additionally has control of a register or can place an additional ROP chain near the SP such that adding or subtracting an offset from the SP is enough to activate this chain. Luckily, in the case of an exploit the machine state is often predictable for an attacker such that these conditions are usually met.

A popular gadget [36] that is often used as a part of a stack pivot sequence is the following[11]:

Listing 2.6: Commonly used stack pivoting gadget in ROP-based exploitation.

```
pop eax ; Load address of control structure into EAX
xchg eax, esp ; Stack Pivot
ret ; Start the execution of the control structure
```

[11] `eax` just serves as an example here. Similarly, the technique could be used with any other register, given that a gadget `xchg <reg>, esp; ret;` exists.

A prerequisite for the use of this gadget is that the address of the control structure is either on top of the stack, placed into `eax` by a previous gadget, or already contained within `eax`. The latter is, for example, the case when a vulnerable function returns a pointer to a data structure that can be controlled by an attacker. Although it seems unlikely that such a data structure address winds up in a register, it is actually a quite common exploitation scenario which is used by techniques such as "ret2reg" (Return to Register) [143]. Unfortunately, however, there is no universal gadget that can be used to pivot the stack. The instruction sequence that is used to accomplish this task always depends on the machine state at the time the exploit is triggered.

Although ROP seems to be quite limited at first glance, it has been shown that the technique can be used to perform Turing complete computations [129]. Given a codebase as large as libc, for example, an attacker can find many different gadgets that enable her to build arbitrary functions by combining the individual blocks in a clever way. This is especially true for the x86 architecture: due to the variable instruction format of the architecture attacker can in this case not only find intended, but also unintended gadgets [138]. However, ROP can be similarly applied to other architectures such as Z80 [21], SPARC [129], PowerPC [86], and ARM [76]. The interested reader can find a more detailed description about ROP in [129].

2.2.1.3 Jump-Oriented Programming (JOP)

ROP relies on the stack and the return instruction to function. The counteract ROP, researchers thus presented various mechanisms that remove return instructions from code entirely (e.g. [82]) or aim to detect ROP based on `ret` instructions (e.g. [23, 38, 39]). The idea of combining small instruction sequences to perform arbitrary computations can, however, not only be realized using return instructions. Instead, this can also be achieved with the help of *indirect jump instructions*. This approach is known as *jump-oriented programming (JOP)* and was first presented by Bletsch et al. [12].

In contrast to ret2libc and ROP, JOP does *not* leverage the SP to control the execution of the gadgets and thus does not require the control structure to be stored on the stack. Instead, the control structure can be located anywhere within memory. This is achieved with the help of a so-called *dispatcher gadget*, which *dispatches* the execution of the *functional gadgets* that an attacker wants to leverage. To make this scheme work, the dispatcher gadget must be invoked at the end of every executed functional gadget. Once invoked, the dispatcher gadget obtains the address of the next functional gadget from the control structure and transfers execution to it. The location of the control structure can thereby be stored within an arbitrary register.

A possible gadget that could be used as dispatcher gadget is the following:

Listing 2.7: Possible dispatcher gadget for JOP.

```
add ebp, edi ; Increase the "IP" located in EBP
jmp [ebp] ; Start the execution of the next gadget
```

In this case the control structure is referenced by the register `ebp`. Whenever the dispatcher gadget is invoked, the value of register `edi` is added to `ebp`, which will essentially point `ebp` to the address of the next functional gadget within the control structure. Once this address has been loaded, the functional gadget is invoked using an indirect jump. The layout of a JOP control structure is shown in Figure 2.7.

While JOP does not rely on the SP and the properties of the return instruction, it places multiple constraints on the functional gadgets. First and foremost, each functional gadget must invoke the dispatcher gadget. This implies that the location of the dispatcher gadget must either be stored in a specific register or must be loaded from memory by the functional gadget. Second, the dispatcher must activate the next functional gadget. To do so it must calculate the location of the next gadget and transfer control to it. This operation usually requires another two registers: one register containing the location of the control structure and one register that is used to calculate the address of the next gadget as shown in Listing 2.7. From this it follows that JOP may require at least three general purpose registers for execution control. These *control registers* must be kept in tact during the entire execution, which implies that none of the functional gadgets can make use of them unless it saves and restores their values. Consequently, the gadgets

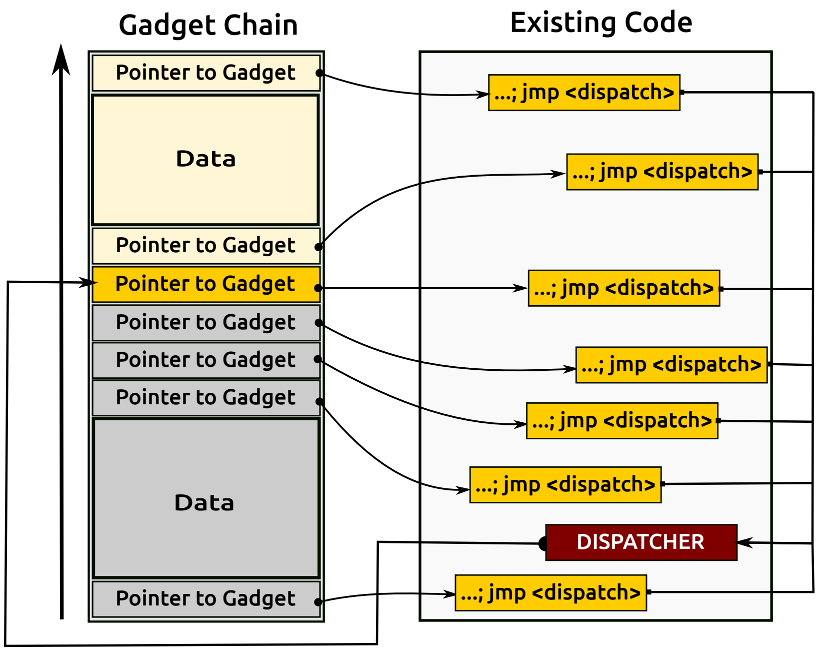


Figure 2.7: The layout of a JOP control structure. The gadgets that have already been executed are shown in grey, while gadgets that will be executed in the future are shown in light yellow. Each gadget ends with a `jmp` instruction that transfers the control to the dispatcher gadget. The dispatcher in turn activates the next gadget. The current gadget is shown in yellow.

that can be used for JOP are much more restricted compared to the gadgets that can be used for ROP. None the less has JOP as ROP been proven to be capable of achieving Turing completeness [12].

2.2.1.4 Sigreturn-Oriented Programming (SROP)

Recently, Bosman and Bos presented a fourth code reuse technique that they refer to as *sigreturn-oriented programming (SROP)* [15]. What is interesting about this technique is that it is, in contrast to the other approaches, based on an OS feature instead of a specific instruction sequence. More precisely, SROP leverages the signal handling of UNIX-like systems (e.g. Linux) as a building block for a code reuse technique. This demonstrates that code reuse mechanisms cannot only be based on specific hardware instructions, but can similarly be implemented using software features. While this may render the technique to be not applicable in general (SROP is OS dependent), it leads to a much larger attack surface and suggests that the problem of code reuse techniques cannot be solved entirely on the instruction level. In addition, attacks based on software features may be much harder to detect compared to attacks relying on specific hardware instructions. To illustrate both of these points consider ROP, for example. From the perspective of the hardware, a return instruction should always return to the address immediately following the last executed call instruction. This is the original purpose of the instruction pair. Consequently, a possible detection mechanisms could verify whether this conditions is met whenever a return instruction is executed. Such an approach that operates on the instruction level can be very effective against *hardware-based* code reuse techniques such as ROP as we will show in Section 7.4.1 where we present such an approach. However, it is most likely powerless against *software-based* code reuse attacks such as SROP, since the property that is used in this case is not residing on a specific instruction, but rather in the *logic of the software.* Signal handling, which is used by SROP, for instance, is an OS system feature provided for applications. To distinguish malicious signals from benign signals, we would thus have to understand the implementation logic behind the signal handler of the application. This is obviously a much harder task than matching `ret` and `call` instructions pairs, which can generally be accomplished without requiring in-depth knowledge of the application.

As previously mentioned, SROP abuses the signal handling of UNIX-like systems. To understand the technique, it is therefore necessary to provide a short overview of the signal handling mechanism found on those systems. We will use Linux as an example for this purpose.

Signals are essentially a way to notify a process that a specific event has occurred. A signal is, for instance, raised when an exception occurs during the execution of a process (e.g. division by zero). Signals are, however, not directly delivered to the process itself, but are first received by the kernel, which handles the signal delivery between processes. To be notified of a signal, a process has to install a *signal handler.* This can, for instance, be accomplished using the `sigaction` system call.

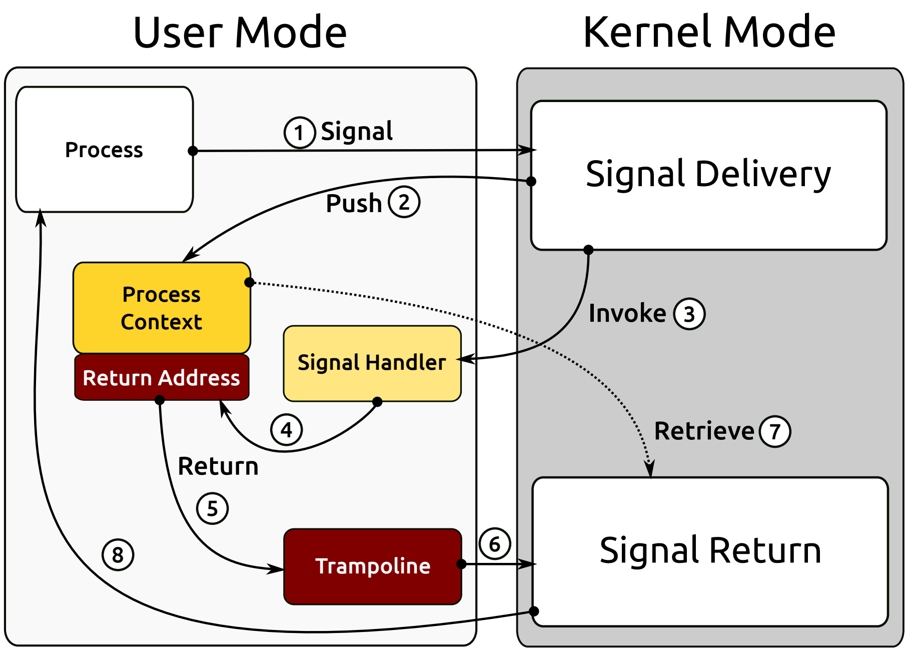


Figure 2.8: Illustration of the signal delivery process on Linux.

When a process has installed a signal handler, the kernel must invoke this handler when the signal associated with it occurs. That is, the kernel must *deliver* the signal. This is essentially a three step process [16]. First, before the signal can be delivered, the kernel must store the current execution context of the process such that it can be restored after the signal handler was executed. Second, the kernel must invoke the signal handler, which is residing within user space, from kernel space. Third, the kernel must regain execution control after the signal handler has been executed to restore the original execution context and to resume the normal execution of the process. Delivering a signal thus requires the kernel to switch multiple times between user mode and kernel mode. This problem arises since the signal handler is located in userland while the signal management is conducted by the kernel. A high-level overview of this process is shown in Figure 2.8.

To save resources and to realize the transitions between kernel space and user space, the kernel manipulates the stack of the process receiving the signal during signal delivery. In particular, it pushes the current execution context and the address of a small execution stub (2) on top of the stack. The latter will thereby function as the return address of the signal handler, which will make sure that the execution stub is invoked, once the signal handler returns (4). Since a user space process cannot directly return to kernel code, the execution stub essentially functions as a trampoline (5) that will return the execution control to the kernel (6). To achieve this the execution stub will invoke the `sigreturn` system call, which will lead to a switch to kernel mode and inform the kernel that signal handling is complete. The kernel will then restore the original execution context using the data that it pushed onto the process stack (7) and finally resume the process (8).

While this is a very lightweight process, it has a crucial flaw: the kernel does not store what signals have been delivered to a process nor does it store the original execution context at a safe location. As a result, an attacker can setup the previously described stack data structures (2) manually and execute a `sigreturn` system call. The kernel will interpret this procedure as a legitimate return from a signal handler (6) and load the attacker controlled execution context (7) before it resumes the process (8), even though it actually never delivered a signal to the process. This enables the attacker to control the entire execution context of the process.

Bosman and Bos [15] propose to use this capability to execute system calls. For this purpose the attacker simply sets up a fake execution context to contain the system call number as well as the corresponding system call arguments, which are all specified in registers on Linux. In addition, she sets the IP within the fake process context to point to a `syscall` instruction, which occur frequently within code. Consequently, as soon as the fake context is loaded, the IP will be set to the `syscall` instruction and the specified system call will be executed.

While execution of a single system call is straightforward, chaining multiple system calls together is more involved. To achieve this the attacker must cleverly combine the properties of multiple system calls. In addition, she must know the address of a `syscall; ret;` gadget. This gadget will effectively function as the glue between our individual system calls similar to the dispatcher gadget in the case of JOP. On a x86-64 architecture the entire approach then works as follows: at first, the attacker sets up a fake signal frame and sets the IP within this frame to point to the `syscall; ret;` gadget. Additionally, she sets the SP to a writable memory address and prepares the other arguments for a `read` system call. Most importantly, this involves setting `rax` to zero, since `rax` specifies the number of the system call according to the Linux ABI and zero is the number of the `read` system call.

As soon as the attacker executes a `sigreturn` system call, the fake frame will be loaded and the `read` system call will be executed. The attacker will use the system call to load attacker controlled data into the memory area pointed to by the SP. When the `read` system call returns, a `ret` instruction will be executed. Since the attacker now controls the data within the stack, she can specify the next gadget. She will use this capability to point the IP once more to the `syscall; ret;` gadget. Consequently, another system call will be executed, but which one?

The register that the attacker used for the `read` system call will all remain the same, except for `rax`, which will contain the bytes read by the `read` system call. As described above `rax` specifies the system call number. Therefore the next system call can be specified by the *number of bytes* that the read system calls returns. Consequently, if the attacker exactly reads 306 bytes, the next system call that will be executed is `sysncfs` [15]. This system call is essentially a no operation (NOP) instruction that will do nothing, but clear the `rax` register. By repeating this scheme the attacker can now execute another `read` system call (`rax` is now zero). This allows her to chain an arbitrary number of `read` system calls together and to read an arbitrary amount of data.

The final piece in the puzzle is how we can execute different system calls. If we read in exactly 15 bytes using a `read` system call, `rax` will contain 15, which is the number of the `sigreturn` system call. By setting up another fake signal frame using one of the previous `read` system calls, we can thus once more make use of SROP to execute an arbitrary system call. This approach enables us to chain arbitrary system calls together leading to a Turing complete language [15].

2.2.2 Code Reuse And Turing Completeness

As stated throughout the last sections, all of the discussed code reuse techniques have been proven to be Turing complete. A crucial aspect that we have not considered so far, however, is the *expressiveness* of these proofs. The problem with proving Turning completeness for a code reuse technique is that the computation abilities of the technique always depend on the gadgets that are available to it. Consequently, a code reuse technique cannot be proven to be Turning complete under all circumstances. Instead, the proof can only be conducted under the assumption of a particular codebase. ROP, for instance, has been proven Turning complete under the assumption that a specific version of the libc is available to it whose gadgets it can leverage as a building block [138]. The expressiveness of the proof is thus limited to this particular version of the libc. A different codebase, even if it is just a different version of the libc, however, is not covered by the proof.

2.2.3 Code Reuse Attacks in Practice

Due to the widespread use of code-based protection mechanisms such as W $\oplus$ X, code reuse techniques have become an integral part of exploit creation. During exploitation they are commonly used to realize the *first* stage of the attack. Hereby the attacker leverages code reuse techniques to create writable and executable memory regions. As a consequence, the following stages of the attack can then again rely on traditional shellcode that is executed from the memory regions created by data-only means.

While we made a clear distinction between the individual code reuse techniques within this section, attackers do not limit themselves to a single code reuse technique in practice. Instead, they use a combination of all techniques to achieve their goals. For example, there is nothing stopping us from combining ret2libc with ROP. Consequently, we do not have to implement every functionality using small instruction sequences, but can reuse functions where available. Similarly, we are not restricted to solely using gadgets conforming to a specific code reuse technique, but can combine ROP, JOP, and SROP to increase the number of gadgets we can leverage and thus the possibilities of our attack.

To account for the fact that code reuse techniques are rarely leveraged in their “pure” from in practice, we will for the remainder of the thesis resolve the artificial boundaries

Technique	Virtual IP	Abused Feature	Control Structure
ret2Libc	FP	`ret` Instruction	Stack
ROP	SP	`ret` Instruction	Stack
JOP	Arbitrary Register	`jmp` Instruction	Anywhere
SROP	SP	`ret` Instruction & Signal Handling	Stack

Table 2.1: Overview of the properties of the code reuse techniques covered within the thesis. As can be seen most code reuse techniques abuse the Stack and the `ret` instruction to function.

between the techniques and will use ROP as representative for all code reuse techniques. For the sake of a better understanding, however, we will try to use "pure" ROP in our examples where possible.

2.2.4 Data-only Programs

According to the Oxford English Dictionary a program is "a series of coded software instructions to control the operation of a computer or other machine" [124]. Since code reuse techniques enable us to create a data structure, the *control structure*, that allows us to *control* the instructions that will be executed by the CPU, this control structure can be seen as the "code" region of a *data-only program* [129]. The *instructions* of this program are essentially *pointers* to machine instruction sequences. Just like a traditional program, the execution of a data-only program requires an IP. Instead of using the IP of the CPU, however, data-only programs leverage a *virtual* IP that points into the control structure. In the case of ROP, for instance, the SP is abused as virtual IP as it specifies which of the pointers within the data-only program will be loaded into the real IP next.

In addition, similar to a normal program, which is, for example, written in C, a data-only program is written in a *code reuse language*. Code reuse languages are created by applying code reuse techniques to a specific set of instructions. Based on this set and the code reuse technique, we can define gadgets that form the basic elements of the language. For instance, given an instruction base such as the libc we can use ROP to define gadgets for loading data from memory, storing data into memory, performing arithmetic operations, and executing conditional jumps, which we can then in turn use to create a Turing complete language [129]. While the instruction set provides the foundation for the language, code reuse techniques provide the concept from which the gadgets and ultimately the language can be derived. If we say a code reuse technique is Turing complete, we thus actually mean that we can create a Turing complete code reuse language based on its underlying concept if a specific codebase is available to us.

Table 2.1 provides an overview of the code reuse techniques covered in this section as well as the virtual IP they leverage, the feature they abuse, and the location where they store their control structure.

2.2.5 Summary

Code reuse techniques provide the basis for data-only malware as they enable us to combine existing instructions into a new program. To accomplish this, code reuse techniques make use of specific hardware or software features that allow them to control the value of the IP based on a data structure. This data structure can be seen as data-only program, whose "code" section consists of pointers to the instruction sequences that should be reused. To start the execution of this data-only program, a code reuse technique dependent virtual IP must be set to its entry point. The virtual IP will then interpret the code of the program and will, with the help of the hardware or software feature that is exploited, sequentially load the pointers into the real IP thus starting the execution of the instruction sequences.

While code reuse is a very powerful approach that can lead to Turing complete languages, the computational ability of a particular code reuse technique always depends on the gadgets that are available to it. We will defer a more detailed discussion of how this limitation affects data-only malware to Section 3.6.1. In addition, we found that most of the existing code reuse techniques rely on the properties of the `ret` instruction to function. This can be exploited to detect code reuse as we will describe in Section 7.4.1.

2.3 Virtual Machine Introspection

While malware formerly operated in user space, we nowadays primarily see malware that attacks the OS kernel [103]. This so-called kernel-level malware operates at the same privilege level as the OS meaning that it can attack and modify any part of the system including the OS kernel itself. Due to this capability such malware poses a very difficult problem for antivirus software. To see this, we have to consider the predominant security model used on most systems today.

Virtually all existing security mechanisms found on a machine are based on the integrity of the OS. That is, the OS is the "foundation of the trusted computing base (TCB) found on most currently deployed computer systems" [115]. Every application running on a system including antivirus software relies on the OS for protection. Once the OS kernel–the heart of the OS–has been compromised, however, this security model is broken and the OS is no longer able to provide any security guarantees. Malware running on the same privilege level as the kernel can perform any action that the OS could perform. There is no longer any *isolation* between a process and kernel-level malware. Consequently, the malware can simply disable any antivirus software running on the system or provide false information to it such that the antivirus software will no longer be able to detect it.

To solve this problem, Garfinkel and Rosenblum [50] proposed the concept of *virtual machine introspection (VMI)*. The key idea behind this concept is to make use of *virtualization* to move security applications out of the machine they try to protect. As a result, the security mechanisms not only regain the property of isolation even in the case of kernel-level malware, but also obtain a *complete* and *untampered* view of the system they try to protect. In the following, we will first provide an overview of virtualization as it serves as the foundation of VMI, before we cover the properties, capabilities, and limitations of the technique in more detail. Once this information has been established, we will then discuss the *semantic gap* problem, which is a fundamental problem that every VMI application faces.

2.3.1 Virtual Machines

One of the key concepts used in computer science to reduce the complexity of a problem is *abstraction*. In the field of software engineering, for instance, we often divide a complex programming problem into classes and subclasses. Each of the classes thereby consists of a set of external functions and a set of internal functions. The external functions provide an *interface* for other classes. This interface essentially forms a level of abstraction. It hides the implementation details of the underlying functions and decouples the classes such that only a minimal dependency remains. Developers can now arbitrarily change the implementation of a class as long as they keep the interface intact. Similar they can make use of other classes without needing to know the details of their implementation. Each class forms its own abstract building block.

In the case of virtualization, we apply the above described concept of abstraction to a software or hardware layer. The fundamental idea behind a *virtual machine (VM)* is to provide a virtual architecture by adding a layer of abstraction to an existing software or hardware interface [147]. The virtual architecture can thereby be fundamentally different from the underlying interface. In particular, a virtual machine can be used to provide a *common* interface even if the underlying interfaces vary greatly. Consider the Java virtual machine (JVM) [85], for example. The JVM provides an abstract computing machine with a well-defined interface on different hardware platforms. This is why Java programs in general can execute on any machine that has the JVM installed.

Since a VM provides an additional abstraction layer to an existing hardware or software layer, it is useful to consider the different interfaces that already exist in today's computer system architectures. This will provide us with a better understanding of the interfaces that a VM can be based on and the interfaces that a VM can export. For this purpose, we will in the following cover the three most important interfaces found within system architectures today.

2.3.1.1 Hardware & Software Interfaces

A simplified view of a system architecture is shown in Figure 2.9. As one can see, there exist three major interfaces. First, there is the interface that is exported by the hardware to the software running on the system. This interface is referred to as the *instruction set architecture (ISA)* (red). The ISA can further be divided into the interface exported

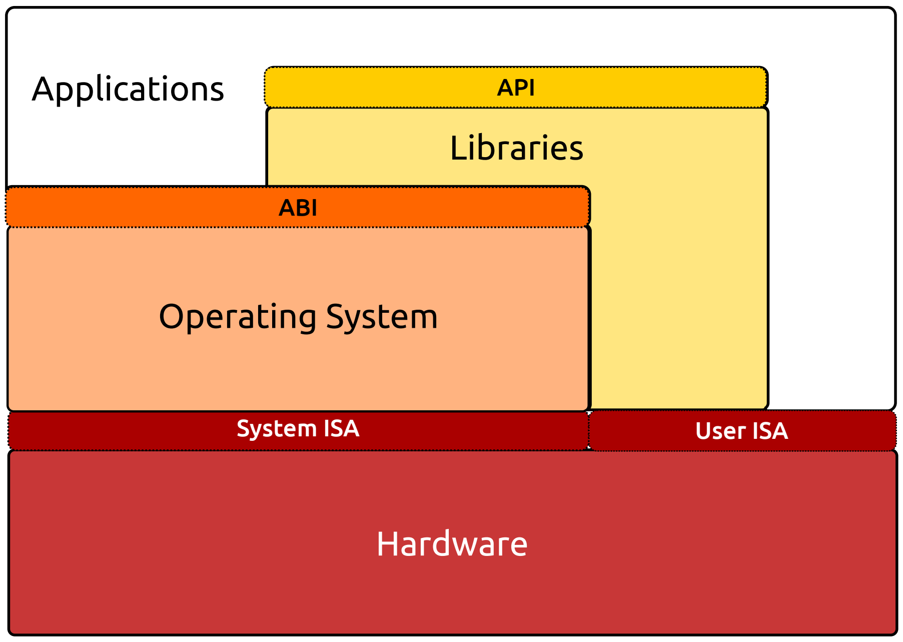


Figure 2.9: The most common interfaces found on today's system architectures. Adapted from [147].

to supervisor software such as the OS (system ISA) and the interface provided to user space applications and libraries (user ISA). The ISA defines the instruction set that the architecture supports. The x86 architecture, for instance, supports the IA-32 ISA.

Second, there is the *application binary interface (ABI)* (orange), which "provides a program with access to the hardware resources and services available in a system" [147]. To provide this functionality the ABI standardizes the interaction between programs on the *binary* level. This information in particular includes the calling convention used and the interaction between user space processes and the OS. The latter is thereby in general accomplished through *system calls*.

With the help of system calls user space application are able to perform privileged operations, which cannot be accomplished using the user ISA alone, such as accessing the hard disc. For this purpose the OS exports a number of functions (system calls) that can be invoked from user space. Once invoked, the OS verifies the access rights of the calling process as well as the arguments provided and then conducts the desired operation on behalf of the process (given that the process is allowed to conduct the requested operation). The details of the communication between the OS and an user space process (e.g. where the system call arguments are stored) are thereby defined within the ABI.

Since the ABI defines an interface on the binary level, it is usually closely bound to a particular ISA. After all, the access to the interface defined by the ABI is always conducted based on the instructions that the underlying ISA supports. This is why an ABI in general cannot exist on its own, but is always defined with respect to some ISA.

To account for this fact, we will for the remainder of the thesis use the abbreviation "ABI/ISA" to refer to an ABI/ISA pair.

Third and finally, it is also common to define interfaces on the *source code* level. These interfaces are referred to as *application programming interface (API)* (yellow). APIs are. for instance, provided by libraries for user space applications and allow them to reuse existing functionality.

Of the three interfaces described above, the ISA and the ABI play an especially important role for virtualization. Based on these interfaces, we can distinguish between two different types of VMs, namely, *process VMs* and *system VMs*. Beginning with the former, we will discuss both VM types in the course of the next sections.

2.3.1.2 Process Virtual Machines

A *process VM* exports an ABI/ISA interface. It enables a process designed for a specific ABI/ISA (OS/hardware) to run on top of the ABI/ISA of the current system. To provide this functionality a process VM exports the ABI/ISA that the target process expects. When invoked, it *translates* the received instructions to the ABI/ISA of the underlying system. Consequently, a process VM provides a virtual environment for a *process*. It is "placed at the ABI interface, on top of the OS/hardware combination" [147].

An example of a process VM is the JVM, which provides an abstract virtual machine as well as an virtual ABI for Java applications. The JVM thereby translates the virtual ABI calls and the instructions of the abstract machine to the ABI/ISA of the underlying system. This enables a Java program to execute as a virtual process on top of the JVM.

In the simplest case, a process VM only provides a virtual ABI, but reuses the ISA of the underlying system, which simplifies the translation process. However, as shown at the example of the JVM, a process VM can also provide its own ISA. In both cases the translation from the virtual ABI/ISA to the real ABI/ISA can be conducted by using one of two *emulation* techniques: *interpretation* or *binary translation* [147]. Interpretation is the most straight forward approach. In this case every instruction (or ABI call) of the virtual machine is emulated using instructions of the real machine. This may, however, lead to a considerable slowdown, since a single virtual instruction may require the execution of many instructions on the real machine.

Binary translation aims to solve this problem by translating entire blocks of virtual instructions, instead of each individual instruction. This enables the *binary translator* to perform optimizations on the block level allowing it to produce instructions blocks that can be efficiently executed on the real ISA. While the optimization process may be slow, the generated instruction blocks can be cached and reused, which will lead to an improved execution time. Thus in comparison to interpretation, binary translation will be slower at the beginning, but its performance will increase over time.

While process VMs have many different applications, they are only of limited use for VMI, since they only allow the execution of user space processes. Therefore they are, for instance, unsuited for the the analysis of kernel space malware. This is why we will not consider process VMs any further within this thesis and will concentrate on *system VMs* instead.

2.3.1.3 System Virtual Machines

In contrast to process VMs, *system* VMs do not provide a ABI/ISA interface, but an *entire system environment* [147]. To achieve this, system VMs directly reside on the system ISA, which allows them to export an arbitrary virtual ISA. As a result, system VMs are not restricted to process virtualization alone, but are able to provide a virtual environment for OSs. The virtualization component providing the virtual ISA is thereby commonly referred to as *hypervisor* or *virtual machine monitor (VMM)*, while a system residing on top of the virtualization layer is referred to as *guest system*, *guest VM*, or simply *guest*. Notice that due to the fact that system VMs provide a system environment, there can be multiple guests running at the same time on the same hypervisor.

System VMs can be differentiated based on the virtual ISA they provide [117]. In particular, a system VM can either provide a *different* ISA than the underlying hardware or the *same* ISA. The former can be achieved with the help of *emulation*. In the process, the system VM emulates the entire hardware operations by mapping the virtual instructions to compatible instructions understood by the real hardware as has been described in the previous section.

On the other side, if a system VM reuses the ISA of the underlying real machine, no emulation is necessary, which increases the performance. To use the approach, however, either the guest OS must support virtualization (this is referred to as *paravirtualization*) or the hardware must provide virtualization extensions that allow the virtualization of an unmodified guest OS. The reason for this is that OSs in general assume to run at the highest privilege level. When running on top of a hypervisor, the guest OS, however, resides at a lower privilege level than the VMM. Due to this fact, the OS will no longer be able to execute specific privileged instructions. If the OS is not aware of the virtualization layer, it will nevertheless try to execute such instructions as it would in case of a non-virtualized environment. This will lead to an exception that may crash the guest system, since the hardware will forbid the execution of privileged instruction for the guest OS, unless the hardware provides virtualization support and will reroute the resulting exceptions to the hypervisor. Since most mainstream processors nowadays provide virtualization support, this is, however, no longer a problem in practice.

In this thesis we will primarily consider same system VMs that make use of hardware support for virtualization. However, most of the techniques that we present can similarly be applied to paravirtualization or different ISA system VMs.

2.3.2 Concept, Capabilities, and Limitations

Virtual machine introspection (VMI) “describes the act of examining, monitoring, and manipulating the state of a guest OS running inside a virtual machine from the isolation of the hypervisor" [117]. Instead of placing security software into the machine it tries to protect, VMI moves security application to the hypervisor level where they can remain functional even if the entire guest system is under the control of an attacker. Due to this approach security applications obtain three essential properties that form the core of VMI [50]:

Isolation This is the most important property from a security standpoint. In contrast to traditional security software, VMI-based security applications will run in isolation from the guest system that they monitor. Software residing in the guest is thereby unable to access or alter security software residing on the hypervisor level.

Interposition To be able to operate, a hypervisor needs to intercept certain events within the execution of the guest (e.g. the execution of privileged instructions). This ability can be leveraged to not only intercept events that are crucial for the functioning of the hypervisor, but also to intercept security critical events within the guest system. Since the hypervisor resides on a layer *below* the guest system, the trapping of an event cannot be disabled or intercepted by the guest. This makes VMI not only interesting for intrusion detection systems (IDSs), but also for intrusion prevention systems (IPSs).

Inspection Even though software running within the guest system cannot access software residing outside of the VM, security software residing on the hypervisor level is able to access and modify the *entire* state of the guest. That is, the isolation property mentioned above only restricts the access of the guest system, while providing the security software with a *complete* and *untainted* view of the guest's state [117]. Since the state of a VM is "compromised of CPU and I/O register values as well as volatile and stable system storage contents" [118], it becomes very difficult for malware to hide from VMI-based security software [50].

While VMI provides three very strong security properties, there are of course limitations to each of those properties that must be considered as well. To provide the reader with a better understanding of the capabilities of VMI and its limitations, we will in the following discuss each of the aforementioned properties in more detail.

2.3.2.1 Isolation

First and foremost the property of isolation does *not* imply *transparency*. That is, a guest system is able to infer that it is running within a virtualized environment. Similarly, malware residing within a guest is able to detect the virtualized environment and may even be able to identify security software running on the hypervisor level. While there has been a lot of work trying to achieve transparency using VMI (e.g. [40, 127, 128]), it is in general believed that achieving true transparency based on virtualization is impossible. For instance, Garfinkel, which is one of the authors that proposed VMI in the first place, claims that "building a transparent VMM is fundamentally infeasible, as well as impractical from a performance and engineering standpoint" [49].

Second, VMI only protects a security application from direct accesses through the guest, it, however, does not protect it against attacks that are a result of the data that is accessed by the security application (e.g. buffer overflows). As a result, VMI-based security applications are susceptible to *indirect* attacks where an attacker manipulates the internal state of the guest OS [5]. When a security application does not expect such manipulations, it may access malformed data which can in turn lead to false-negatives or in the worst case the triggering of a vulnerability in the security application. Consequently, any data read from the the guest's state should be considered malicious and must be handled with care.

Finally, isolation can of course only be provided given that the hypervisor was implemented correctly and does not contain any vulnerability. Practice has shown, however, that hypervisors–similarly to any other software–can contain exploitable software bugs that allow to compromise the hypervisor [45, 105]. In this thesis, we consider such attacks out-of-scope and assume that the hypervisor has been implemented correctly. In practice, the hypervisor can be protected against attacks using mechanisms such as HyperSafe [173].

2.3.2.2 Interposition

While different ISA system VMs can in general intercept arbitrary events within the guest due to emulation, the same does not apply for same ISA system VMs. To intercept a hardware event from the hypervisor in this case, one must cause a VM exit whenever the hardware event occurs. This process is also referred to as *trapping* an event. In many cases, trapping an event can be easily accomplished with the help of the hardware itself, as in the case of page-faults or control register changes. However, considering the total number of hardware events that exist, the events that can be natively trapped still only form a small subset. It is therefore a common challenge with VMI-based applications to find a mechanisms to trap events not directly supported by the hardware.

In general, the approach in such a situation is to generate a secondary hardware event for which trapping is supported, whenever the desired hardware event occurs. As an example, consider the interrupt trapping mechanism that is used by Nitro [120]. In this case, the authors wanted to trap every occurrence of a user-defined interrupt (which are the interrupts 33-255 in the interrupt descriptor table (IDT)) to the hypervisor on the x86 architecture - an event that is not natively supported by the hardware. They accomplished this by setting the IDT limit, which contains the size of the IDT, to only include the first 32 entries. As a consequence, every interrupt with a number higher than 32 causes a general protection fault, a trappable event.

Other researchers have shown that the same or a similar signal can be used to trap code execution [112] (page faults), individual instruction types such as `call` or `ret` instructions [168] (non-maskable interrupts), or sysenter/syscall instructions [40] (general protection fault). Besides using a signal, it is also possible to use specific instructions such as `HLT` [169] to trap events. In this case, the event trapping mechanism must ensure that a trappable instruction is executed directly after the event occurred.

To make use of an event trapping mechanism in a VMI application, the mechanism must not only reliably trap the event of interest, but it must also provide all of the properties that the application requires. While the most important properties of an event trapping mechanism are certainly tamper-resistance and stealth, performance naturally is also essential. To achieve good performance results, the event trapping mechanism should only produce VM exits when the desired event occurs. Therefore events that frequently occur during normal operation such as page faults should in general be avoided for event trapping if possible.

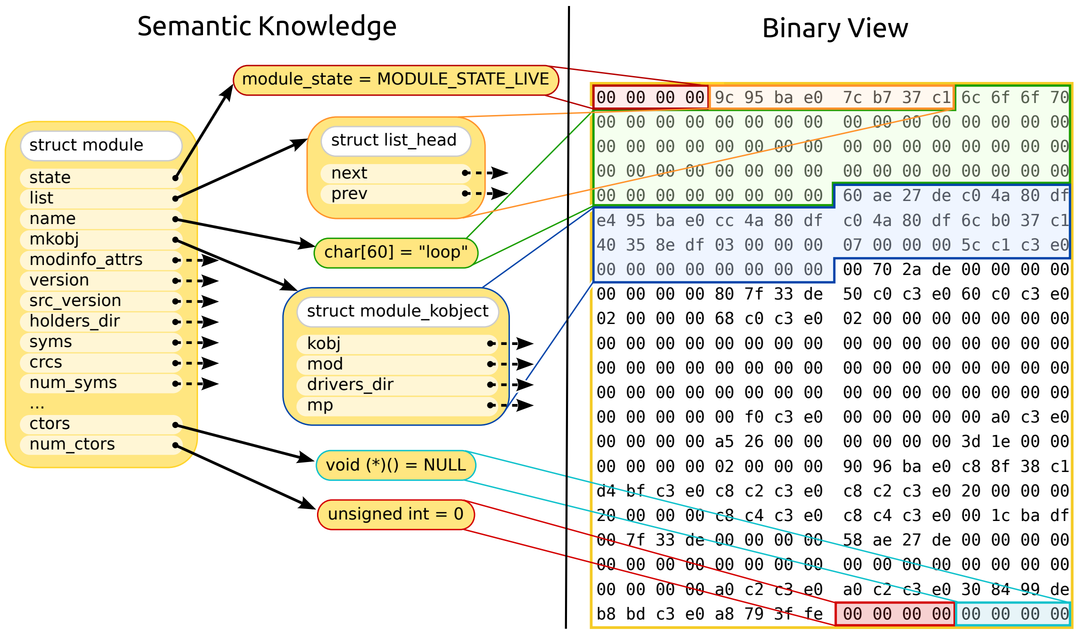
2.3.2.3 Inspection

While an VMI-based security application has access to the entire state of the guest system, making use of this state is not as straight forward as one might think. To effectively use the state one has to overcome one of the fundamental problems of VMI, which is referred to as the *semantic gap* [22]. Due to the significance of the problem, we will cover it in a section of its own.

2.3.3 The Semantic Gap

Security applications residing on the hypervisor level are able to inspect the entire state of a guest system. However, their view of this state is very different from the view that the guest OS has. While the view of the latter is compromised of high-level data structures, the view of VMI-based security applications is restricted to the bits and bytes contained in the guest's state. This is due to the fact that the security application lacks the *semantic knowledge* of the guest OS that is required to interpret the binary, low-level state of the guest correctly. This semantic disconnect between the hypervisor and the guest OS is referred as the *semantic gap* [22].

The semantic gap is illustrated in Figure 2.10. As can be seen, the semantic knowledge of the guest OS is crucial to be actually able to make use of the state that is available to the hypervisor. Without this knowledge, a VMI-based security application is limited to the "binary view" of the state which contains all information, but is nearly useless without semantics. To solve this problem, we have to *generate* a view on the hypervisor level that is similar to the view the guest OS has and allows a VMI-based security application to access any information that it requires to function. This process is referred to as *bridging* the semantic gap. Pfoh, Schneider, and Eckert [118] presented three general approaches that can be used to accomplish this: *in-band delivery*, *out-of-band delivery*,



```
00 00 00 00 9c 95 ba e0  7c b7 37 c1 6c 6f 6f 70
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  60 ae 27 de c0 4a 80 df
e4 95 ba e0 cc 4a 80 df  c0 4a 80 df 6c b0 37 c1
40 35 8e df 03 00 00 00  07 00 00 00 5c c1 c3 e0
00 00 00 00 00 00 00 00  00 70 2a de 00 00 00 00
00 00 00 00 80 7f 33 de  50 c0 c3 e0 60 c0 c3 e0
02 00 00 00 68 c0 c3 e0  02 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 f0 c3 e0  00 00 00 00 00 a0 c3 e0
00 00 00 00 a5 26 00 00  00 00 00 00 3d 1e 00 00
00 00 00 00 02 00 00 00  90 96 ba e0 c8 8f 38 c1
d4 bf c3 e0 c8 c2 c3 e0  c8 c2 c3 e0 20 00 00 00
20 00 00 00 c8 c4 c3 e0  c8 c4 c3 e0 00 1c ba df
00 7f 33 de 00 00 00 00  58 ae 27 de 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 a0 c2 c3 e0  a0 c2 c3 e0 30 84 99 de
b8 bd c3 e0 a8 79 3f fe  00 00 00 00 00 00 00 00
```

Figure 2.10: The figure illustrates the semantic gap based on an excerpt of the guest's physical memory. On the right the "binary view" is shown. This is the view of the hypervisor. The guest OS can make use of the semantic knowledge shown on the left to interpret this binary view. This information, however, is not available to the hypervisor. (Source: [133]).

and *derivation*. Before we can describe these approaches in more detail, however, we must first introduce the properties that we will use to compare and contrast them.

2.3.3.1 Properties

In this section we will introduce important properties that must be taken into account when creating a solution that bridges the semantic gap, which we from here on simply refer to as a *bridging component*. We will use these properties in the following sections to provide the reader with a better understanding of the advantages and disadvantages of the individual available approaches that try to accomplish this task.

Binding. Bridging components often make *assumptions* about the guest system based on *their* semantic knowledge [117]. For instance, they assume that the guest OS provides a specific interface or a specific data structure layout. The internal implementation of a software interface or a data structures may, however, change. If the assumptions of the bridging component are *not bound* to the software state of the guest, it may not be aware of such changes and may rely on false assumptions. For example, to obtain the list of processes running within a guest system, a bridging component may try to access the guest's process list that it assumes to be stored at a specific location. There is, however, no guarantee that this list is actually used by the guest. Instead an attacker could have altered the software state of the system such that it uses a different hidden process list. That is, an attacker can invalidate the assumptions made by the bridging component to present a seemingly good system state to a security application, while the system is compromised in reality.

Such an attack is possible, since the assumptions made by the bridging component are *dependent* on the software state of the guest, while the state of the guest is *independent* of those assumptions. Consequently, we refer to such an approach as being *non-binding* [87], since the software state of the guest is not bound to the assumptions of the bridging component. On the other side, we call an approach *binding* if an attacker is unable to change the software state of the inspected guest in a way that invalidates the assumptions of the approach without the bridging component being aware of the attack.

Guest OS Portability. A bridging approach is guest OS portable [118], if it is independent from the guest OS used. In this case, the bridging component can be used for different OSs *without* changing its implementation. Otherwise, if an approach is not guest OS portable, each guest OS may require its own implementation of the bridging component.

Hardware Portability. Similar to being independent of the guest OS, an approach can also be independent of the underlying hardware. In this case the approach can be applied to multiple hardware platforms without changing its implementation. We call such an approach *hardware portable* [118].

Full State Reconstruction. Not every bridging approach may be able to reconstruct the entire high-level state of the the guest system on the hypervisor level [117]. While this feature may not be required for every VMI-based security application, the more complete the reconstruction of the state is the better suited is a bridging component in general as a foundation for VMI-based security applications. We say an approach is able to achieve *full state reconstruction* if it provides access to the *same* information as the guest OS.

Stealth. Stealth specifies how easy it is to detect a VMI-based security mechanism from within the guest system. Although VMI-based mechanisms are not transparent as has been discussed in Section 2.3.2, the degree of stealth that an approach provides is still important. While virtualization is in most cases easily detectable, inferring which security applications reside on the hypervisor level can be a very challenging task especially for mechanisms with a high level of stealth. Since an attacker may decide to infect a system based on the present security mechanisms (e.g. to avoid the infection of a honeynet), it is crucial that security applications do not easily give away their presence. We consider a VMI-based mechanism to achieve a high level of stealth, if it cannot be detected *directly* from within the guest system by reading some software or hardware value (e.g. register), but only based on some *side channel* such as *timing attacks*.

Isolation. Naturally isolation is one of the most important aspects for a mechanism that bridges the semantic gap from a security perspective. Ideally, the bridging mechanism should have the exact same isolation properties as the hypervisor. For a more detailed discussion of this property we refer the reader to Section 2.3.2.1.

2.3.3.2 In-Band Delivery

An in-band delivery approach takes advantage of the guest system itself to gather high-level state-information. This is accomplished by placing an *agent* into the guest system. Upon request, this in-guest agent can then simply query the guest OS for the desired information and transfer it to the hypervisor. Instead of bridging the semantic gap, this approach thus rather *circumvents* the semantic gap.

In-guest approaches tend to have a better performance than out-of-guest approaches as they reduce the number of VM exists, which are in general very costly operations [141]. However, with in-band delivery, one inadvertently trades stealth and isolation for inspection and performance as the agent will execute inside the guest OS. Since the guest system may be compromised, it seems like we do not gain anything from this approach compared to traditional anti-virus software. In both cases a component resides within the system. This component can be attacked and disabled by malware running in the machine. However, there is a major difference between the scenarios: in the case of an in-guest component we can make use of a trusted entity, the hypervisor, to protect the component from tampering. This option is not available in a non-virtualized scenario. Therefore the main challenge for an in-band delivery approach from a security perspective is to protect the in-guest component during its execution in an untrusted guest system from the hypervisor.

In addition, an in-band delivery approach is in general non-binding, since it relies on the information provided by the guest. There is, however, no universal way for the agent to decide whether the received information reflects the real state of the guest or if was manipulated by an attacker. Clearly, an in-guest approach is also not guest OS portable. An agent designed for Windows, for instance, cannot be used to retrieve information from within a Linux guest. Instead, each guest OS requires its own individual in-guest agent. An in-band approach is, however, hardware portable and able to achieve full state reconstruction.

We will cover existing in-guest approaches in more detail in Section 6.5.

2.3.3.3 Out-of-Band Delivery

In contrast to in-band delivery approaches, out-of-band delivery approaches do not rely on an in-guest component to bridge the semantic gap. Instead, they try to solve the problem entirely from the VMM by reconstructing the guest's data structures on the hypervisor level using semantic knowledge that is delivered to the view generating component in advance. To obtain this knowledge these approaches make use of information sources such as exported kernel symbols [66], debugging symbols [50], static source code analysis [19, 133, 134], signatures for kernel data structures [42], or program traces [41].

Although out-of-band approaches are able to narrow the semantic gap, full state reconstruction has, to the best of our knowledge, not been achieved so far. One of the main reasons for this is certainly the complexity of modern OS kernels, which makes it very difficult to recreate the entire view of the guest OS on the hypervisor level. In addition, out-of-band approaches are not guest OS portable and non-binding [117, 133]. The latter being a result of the fact that the necessary semantic knowledge used for reconstruction is delivered to the bridging component beforehand using an out-of-band channel. Consequently, this knowledge is not bound to the current software state of the guest. On the plus side, out-of-band delivery approaches, however, are in general hardware portable, and provide a high-level of stealth, since they operate entirely on the hypervisor level out of the reach of the guest.

2.3.3.4 Derivation

Derivation fundamentally differs from delivery-based approaches in that it does not use software-based semantic knowledge to bridge the semantic gap, but instead leverages semantic knowledge that can be derived from the (virtual) hardware to accomplish the same. By using a hardware centric approach, derivation is the only bridging technique that can provide guest OS portability and binding. Since the semantic knowledge used by the approach is based on the hardware, an attacker cannot simply alter the software state of the guest system to invalidate this knowledge. Instead, the attacker is forced to act according to the rules that the hardware lays upon her. In addition, derivation often achieves a high-level of stealth given that the bridging component does not alter the virtual hardware in way that can be observed by the guest.

However, this functionality comes at a cost. First of all, derivation is not hardware portable. Second and more importantly, the semantic knowledge that can be derived from the hardware is quite constrained [117]. Consequently, hardware portable approaches are far from being able to achieve full state reconstruction and must in practice often be combined with delivery-based approaches to tap their full potential. Examples of derivate approaches include Patagonix [87], Lycosid [67] and Nitro [120].

2.3.3.5 Comparison of the Approaches

Table 2.2 summarizes the properties of the individual approaches to bridging the semantic gap. Derivation is the only approach that is binding and guest OS portable. However, the approach is also the only one that is not hardware portable and unable to reconstruct the full state. Both derivation and out-of-band delivery achieve a high-level of stealth as

Property	*in-band*	*out-of-band*	*Derivation*
Binding	✗	✗	✓
Hardware portable	✓	✓	✗
Guest OS portable	✗	✗	✓
Full State Reconstruction	✓	✓	✗
Stealth	✗	✓	✓
Isolation	✗	✓	✓

Table 2.2: Comparison of the different approaches to bridging the semantic gap.

they operate outside of the guest system. In addition, out-of-band delivery is hardware portable and can in theory achieve full state reconstruction. To the best of our knowledge, this has, however, not been accomplished so far. In-band delivery approaches on the other side naturally achieve full state reconstruction as they operate within the guest system. Due to this property they are in contrast to the other approaches unable to achieve a high-level of stealth and isolation though. Just as out-of-band delivery approaches they are hardware portable, but non-binding.

2.3.4 Summary

The security of most systems today is based on the integrity of the OS. The OS thus presents a single point of failure. Once the OS is compromised all security is lost. VMI provides a solution to this dilemma by leveraging full hardware virtualization to move security applications out of the system they try to protect. As a consequence, security applications can remain functional even if the entire VM including the OS is compromised. This capability comes at the cost of the semantic gap though, which describes the semantic disconnect between the hypervisor and the guest OS. To solve this problem, we must generate the semantic view of the guest OS on the hypervisor level. This can be accomplished by placing an agent into the guest (in-band delivery), by providing the necessary semantic knowledge to the hypervisor in advance (out-of-band delivery), or by deriving information from the hardware (derivation).

2.4 Malware Detection & Prevention

In this thesis we are not only concerned with the capabilities and limitations of data-only malware, but also want to research how we can mitigate this novel threat should if prove dangerous. That is, we want to consider data-only malware from the vantage point of the attacker as well as the defender. In this context one of the important issues that we have to analyze is the effectiveness of existing countermeasures against data-only malware.

To be able to do so, however, we must first provide the reader with an overview of the malware prevention and detection mechanisms that can be found on systems today.

2.4.1 Protection Mechanisms

Modern systems employ a variety of software and hardware mechanisms to hinder the successful execution of malware. In the following, we provide an overview of the mechanisms that can commonly be found on x86 Linux and Windows systems.

2.4.1.1 Software-based Mechanisms

StackGuard. One of the first approaches to counter buffer overflow attacks was StackGuard [33]. StackGuard places a new field (*canary*) between a function's local variables and saved return address on the stack. When initializing a function's local stack, the OS automatically initializes the canary with a random value whose integrity is verified before returning. The general idea behind this approach is that any buffer overflow attack that overwrites the return address must also overwrite the canary, and since this canary is initialized with a random number, it is very difficult for an attacker to guess the correct value. Based on this, the program can determine that if the canary's value changed a buffer overflow likely occurred and can take appropriate actions (e. g., terminate).

Kernel Patch Protection (PatchGuard). Beginning with the 64-bit version of Windows XP, Microsoft introduced a mechanism that runs at regular intervals and verifies the integrity of the kernel's code sections and important kernel data structures called PatchGuard [72]. In particular, PatchGuard attempts to verify the integrity of those portions of the kernel that are often patched by a rootkit or other malware such as its code sections, system service tables, descriptor tables, etc.

Address Space Layout Randomization (ASLR). Another contemporary protection mechanism is address space layout randomization (ASLR) and its kernel equivalent kernel ASLR (KASLR) [51]. The main idea behind this approach is to randomize the base address where code sections are loaded to. This makes it difficult to employ exploits that make use of existing code. In fact, ASLR is supposed to act as the hard counter to code reuse attacks such as ROP and ret2libc. Due to the fact that code reuse attacks makes use of existing code snippets, these attacks are hindered when those code snippets are loaded at random offsets. This leaves the attacker in a situation in which she is forced to guess the location of gadgets or functions.

2.4.1.2 Hardware-based Mechanisms

W $\oplus$ X. One of the oldest protection mechanisms is $W \oplus X$. This approach makes use of the paging (or segmentation) features of particular hardware (e. g., x86). Such features often allow a level of read/write/execute access control at page or segment granularity. The $W \oplus X$ mechanism works by marking single pages as either writable or executable, but never as both simultaneously. This prohibits an attacker from introducing new code as "data", then manipulating the system to execute that code. It also prohibits an attacker from directly introducing new code in those areas of memory reserved for code as they are not writable. That is, memory is split up to contain either code or data and the code sections can not be written to while the data sections cannot be executed.

Supervisor Mode Execution Prevention (SMEP). A fairly new protection mechanism introduced by Intel is SMEP [64]. When this feature is enabled, the processor will fault when the current privilege level (CPL) of the processor is less than three and an attempt to execute code from a page whose supervisor bit is not set is made. This means that the processor will not allow the execution of code in user space while operating in kernel mode. This is useful against attacks in which code is loaded into user space –which requires no special privileges– and the kernel control flow is manipulated into jumping to this code segment.

Supervisor Mode Access Prevention (SMAP). As the name suggests, supervisior mode access prevention (SMAP) is closely related to SMEP. While SMEP is focused on code execution, however, SMAP prevents accesses to user data from kernel code. In particular, when enabled the processor will fault whenever an instruction is executed at CPL less than three which tries to access data stored within a page that does have its supervisor bit set. This prohibits the kernel from accessing user data and forces an attacker to find a way to load malicious data into kernel space before she can use it for code reuse attacks such as ROP. Since the kernel sometimes requires access to user data (e.g. to execute a system call), the caveat of the technique is that the kernel must disable SMAP whenever such an access is required and re-enable it after the access was made. For this purpose Intel introduced two new instructions `CLAC` and `STAC` [111].

Code Signing. In contrast to the other mechanisms described so far which generally aim to prevent dynamic exploits, code signing approaches work by validating the integrity of the code while it is loaded. This is accomplished by leveraging digitally signed binaries that are checked before loading and are only loaded if the binary is unchanged and signed with the key of a trusted party. As such an approach is quite restrictive, it is generally used in kernel protection rather than in userland protection.

With the assistance of hardware, code signing can be used to implement a trusted boot sequence. This works such that the boot ROM (the root of trust embedded in the hardware) only loads an untainted and signed bootloader, this bootloader only loads an untainted and signed kernel, and finally the kernel only loads untainted and signed drivers or modules. By building such a chain of trust and rooting it in hardware one can be very certain that all code that is loaded into the kernel is untainted and trusted *at* *the time it is loaded.* The UEFI specification describes such a mechanism and is used in modern PCs [163]. However it is important to note that this will not prevent code from being introduced at runtime through a vulnerability, for example.

2.4.2 Malware Detection

Having provided an overview of current software-based and hardware-based protection mechanisms, we now cover present malware detection mechanisms. Instead of focusing on individual detection approaches, however, which would due to their sheer number go far beyond the scope of this thesis, we discuss the major *detection concepts* leveraged today.

While the individual definitions may vary, there exist two fundamental malware detection concepts in literature: *signature-based* detection and *anomaly-based* detection [63, 110]. In addition, many authors also consider a third category that is particularly important for data-only malware: *integrity-based* detection [158]. In the following we will consider each of these categories in turn. In the process, we will also discuss *hook-based* detection, which is a specific form of integrity-based detection that is specifically relevant for our analysis. Before we begin with our discussion, however, we define important terms that we are going to use throughout the section.

2.4.2.1 Definitions

There are two types of data that we will encounter during our discussions in this section: *control data* and *non-control data.* The former can thereby further be divided into *transient* and *persistent* control data. In the following we will define these types of data.

Control data and non-control data. *Control data* specifies the target location of a branch instruction. By changing control data, an attacker can arbitrarily change the control flow of an application. Examples of control data are return addresses and function pointers.

In contrast, *non-control data* never contains the target address for a control transfer. In certain cases, however, it may influence the control flow of an application. For instance, a conditional branch may depend on the value of non-control data.

Transient and persistent control data. We consider control data to be *transient* when it cannot be reached through a pointer-chain originating from a global variable. This essentially implies that there is no lasting connection between the application and the control data. Instead, the control data is only visible in the current scope of the execution such as a return address which is only valid as long as a function executes.

By extension, we consider all control data that is reachable through a global variable as *persistent*, since the control data is permanently connected to the application and can thus always be accessed independent of the current scope.

2.4.2.2 Signature-based Detection

One of the most well-known and widely used malware detection mechanisms is *signature-based* detection. The main idea behind this approach is to calculate an unique digital fingerprint (signature) for each known malware instance, which can then be leveraged to identify and detect malware reliably. In general, a signature is thereby essentially a regular expression that matches a specific byte pattern within the malware binary. The detection process then consists of applying the regular expression to each file that should be checked for a malware infection. If a regular expression created for a malware binary matches on a file, the file is considered to be an instance of the malware the signature belongs to.

To make this scheme work, it is essential that each pattern that is used to identify a malware instance is unique. To this end researchers presented various mechanisms that are capable of creating signatures for malware detection automatically (e.g. [53, 83, 137])[12] . However, since existing approaches still suffer from false positives and may not be able to generate reliable signatures for all malware types (especially when the malware is obfuscated), signature creation primarily remains a manual process [63].

The main disadvantage of signature-based detection is that the approach is in general only able to detect *known* malware. After all, one must calculate a signature of a malware sample before it can be used for detection. In addition, malware may be able to *evade* signature-based detection using techniques such as polymorphism or metamorphism (see Section 2.1.4 for details). To counteract such approaches, anti-virus scanners usually not only scan binaries before their execution, but emulate the execution of the binary [157] or scan the memory of the system for signatures. As a result, the anti-virus becomes able to detect malware that unpacks or decrypts itself in memory. This approach, however, can only be used against polymorphic malware, but is insufficient for metamorphic malware.

To detect metamorphic malware and to improve signature-based detection, researchers proposed to make use of *semantic* signatures [28, 78]. Instead of calculating a *syntactic* signature based on the binary representation of the malware, semantic signatures try to capture semantic properties of the malware. This can, for instance, be achieved with the help of *templates*. While a syntactic signature is often based on concrete instructions, the semantic template-based signature will specify the effect of the instructions using symbolic values. As a result, the signature will match as long as the instructions executed by the malware will lead to the same result as the symbolic calculation specified in the template. Since metamorphic malware typically employs syntactic metamorphism, while keeping the semantic of the malware intact, such an approach can be very effective. However, Moser, Kruegel, and Kirda [97] have shown that semantic signatures are not foolproof either and can be evaded as well.

While signature-based detection is vulnerable to evasion attacks, the approach also has a significant advantage: in general signature-based detection produces almost no false positives, which is a very important property in practice. In fact, detection accuracy is

[12]The interested reader can overview of such approaches in [71].

in many cases almost as important as the detection rate, which may be one of the reason why signature-based detection–in spite of its drawback–still remains widely in use.

2.4.2.3 Anomaly-based Detection

The key idea behind anomaly-based detection is to detect malware based on the behavior it exhibits. More precisely, we assume that the behavior of a system infected with malware will deviate significantly from the behavior the system would normally have. Since the behavior of malware is independent of its implementation, this approach is capable of detecting known as well as unknown malware.

To detect anomalies, we first create a model of normal behavior of a user, a program, a system, a network etc. In the next step, we monitor the entity for which the model was created. The runtime data it generates is then compared to the created model. If the entity conforms to the model, it exhibits normal or legal behavior. On the other side, if the entity deviates from the model, it exhibits abnormal behavior. Since malware is expected to modify the behavior of a system from normal to abnormal, the latter is interpreted as a sign of a malware infection.

Naturally, the effectiveness of anomaly-based detection stands and falls with the quality of the created model that describes the normal system behavior. In general, there exist three different techniques to create such a model: *statistical anomaly detection*, *machine learning based anomaly detection*, and *data mining based anomaly detection* [110]. Each of these techniques thereby makes use of a *training phase* and a *testing phase*. During the first phase, the training phase, the model of normal behavior is generated. Later on this model is then used in the testing phase to detect abnormal behavior.

In the case of statistical anomaly detection, the detector requires two different profiles for detection. Namely, the *stored* and *current* profile. The stored profile contains the model of normal behavior created in the training phase. It is obtained by monitoring the target entity and collecting statistics about its behavior. For example, one could record how much network traffic and CPU usage a process has. During the testing phase the stored profile is then compared to the current profile, which reflects the behavior that the monitored entity currently exhibits. For this purpose the current profile is constantly updated. The deviation between the stored profile and the current profile is then used to calculate an anomaly score. Should the anomaly score reach a predefined threshold, the detector will consider this as abnormal behavior and raise an alarm.

In contrast to statistical anomaly detection which uses predefined features and thresholds, machine learning based anomaly detection leverages machine learning to improve its performance based on previously obtained results. For this purpose, the detector receives a training set that it uses to “learn” a model of normal behavior. Based on this model it then classifies the currently observed behavior as either normal or abnormal.

To use statistical and machine learning based anomaly detection approaches we have to select features that allow us to distinguish normal from abnormal behavior. This often requires the involvement of an human expert. However, modern systems contain so many features (e.g. CPU, disc, or network usage, system and API calls, running process etc.) that some patterns within data may even remain hidden from an expert. Consequently, we may miss the features that may actually be suited best for detecting the anomaly we are looking for.

To solve this problem, researchers proposed the concept of data mining based anomaly detection. The main idea thereby is to automatically extract patterns of normal (or malicious) behavior from large amounts of audit data that can then be used for anomaly detection [80]. For this purpose data mining techniques are applied to the audit data to discover the necessary knowledge. Once a pattern has been found, it can then be used to detect anomalies.

The main issue of anomaly-based detection is that it is based on probabilities. In contrast to signatures, behavior is not a concrete value that can be measured and compared. As a result, normal system behavior may be classified as malicious (false positive) or malicious behavior may be classified as normal (false negative). Thus while anomaly-based detection is capable of detecting unknown attacks, it is not as reliable as signature-based detection.

2.4.2.4 Integrity-based Detection

Another popular approach to malware detection is to detect malware infections based on *integrity violations* [158]. The approach makes use of the observation that malware must modify the state of the system it infects to function. To see this, one must only consider the malware taxonomy introduced by Rutkowska [131] (see Section 2.1.4.4 for details). In her taxonomy Rutkowska classifies malware based on the *type of modification* that the malware conducts to the system state. Consequently, we can detect malware infections by identifying malicious changes to the system state, which is the basic idea behind integrity-based detection.

The complexity of the approach thereby heavily depends on the part of the system state whose integrity should be validated. As observed by Rutkowska, the system state can be roughly divided into constant and dynamic parts. The validation of constant parts of the system state is thereby generally easy: to detect malware infections, it is sufficient to detect *any* change within the monitored region, since a constant part should *never* change during normal operation. Validating the integrity of dynamic regions on the other side is often difficult. Instead of simply detecting changes, we must in this case determine whether a change conducted to the dynamic part of the system state was malicious or benign. Depending on the part of the state that we target, it may thereby become quite challenging to find integrity constraints that are restrictive enough to detect malware modifications, but are at the same time loose enough to avoid false positives.

Existing detection methods that fall into this category can be classified into *file* integrity checkers, *code* integrity checkers, *control-flow* integrity checkers, and *data* integrity checkers. In the following, we consider each of the mechanisms in turn.

File Integrity Checkers. Viruses infect files by appending the virus body to them. By doing so, they modify the file. Since binaries are in general static, this behavior lends itself well for a detection mechanism. The main idea thereby is to calculate a cryptographic checksum of every binary in a secure environment. Once these checksums have been created, they can be used to verify whether a file has changed by recalculating the cryptographic checksum of the file and comparing its value to the original checksum. If the checksums differ, the file was changed.

Code Integrity Checkers. Another important part of the system state is the *codebase*. Since traditional malware consists of executable instructions, it *must* add additional instructions to the system or modify existing instructions to function. That is, traditional malware is *forced* to change the codebase of the system to run. Because code regions of programs are usually static once the program has been loaded into memory, this property is naturally predestined for the detection of traditional malware as it is common to *all* malware types and on top of that relatively easy to implement. In fact, we can leverage a similar approach as in the case of file integrity checking and calculate cryptographic checksums (hashes) for code regions, which can then be validated using a whitelist.

To protect the component responsible for code integrity validation, code integrity checker in general either rely on specific hardware (e.g. [114]) or leverage virtualization (e.g [116, 128]), which is the more common approach. The latter has the additional advantage that the whitelist used for comparison can be stored on the hypervisor level as well such that it is isolated from the monitored system.

When it comes to the validation process itself, the main issue that must be considered is that code regions are only static once they have been loaded into memory. During the loading process, however, the loader may modify code regions to prepare them for their execution. For example, may the loader have to update addresses used within the code when ASLR is leveraged and the base address of the code section is randomized. As a result, the final hash value of a code region is dependent on the modifications applied to a binary at load time.

The general approach to handle this issue is to make use of a *trusted store* that contains all benign binaries instead of a plain whitelist only containing the hashes of the benign code regions. During the validation of a code region, the integrity checker then simulates the changes conducted by the loader on the benign binary and compares the result with the code region in memory. A similar approach is for example used by Patagonix [87], a well-known a hypervisor-based code integrity system. Instead of simulating the changes, Patagonix, however, reverts the changes conducted by the loader to obtain the original code section as it is stored within the binary. Once the original layout of a code region has been restored, its hash can be calculated and compared to the whitelist.

Control Flow Integrity Checkers. The general goal of an attacker is to execute her own code. To accomplish this she must modify the *control flow* of the system at some point in time and redirect its execution to her code. Abadi et al. [1] proposed to leverage this deviation of the normal control flow for a detection mechanism which they refer to as control flow integrity validation (CFI). The idea behind CFI is to ensure the integrity of the control flow of an application by constructing its control flow graph (CFG) and validating that every control transfer of the application follows the rules of this graph. Since the control flow modification of the attacker will in general not correspond to the CFG of the application, this results in a protection mechanism that is difficult to evade [2]. The enforcement of the control flow validation can thereby be either conducted by inserting control flow checks through recompilation (e.g. [81]), by instrumentation (e.g. [186]), using virtualization (e.g. [34]), or by using specific hardware features such as the performance counters (e.g. [181]).

Data Integrity Checkers. Validating the integrity of data is in general a lot more complex than validating the integrity of code or files. While the latter are usually invariable, data is dynamic and may constantly change. As a result, detecting modifications alone is no longer sufficient to detect malware infections. Instead, the changes conducted to a data structure must be validated. How this can be accomplished in practice depends on the data structure as well as the application that uses it. That is, data integrity validation is in general application specific.

Existing data-integrity checkers are primarily focused on the OS kernel. Petroni et al. [115] were the first to propose a general architecture for the detection of kernel data integrity violations. Since then various systems have been proposed that try to detect or prevent malicious modification of kernel data structures [19, 57, 79, 127, 134]. However, what is common to all these approaches is that they only *enforce* integrity checks, but leave the *creation* of the actual integrity constraints to a human expert. To the best of our knowledge, the only approach that tries to generate integrity constraints for kernel data structures automatically is Gibraltar [7]. While this approach provides a good starting point and could support a human expert in the creation of integrity constraints, the authors acknowledge that the generated invariants are “neither sound nor complete” [7].

2.4.2.5 Hook-based Detection

The final detection approach that we want to consider is *hook-based* detection. On close examination hook detection is actually a form of integrity-based detection. However, due to the importance of the method, we cover it in a subsection of its own.

The fundamental idea behind hook detection is that malware must in general intercept events within the system to able to fulfill its purpose as has been described in Section 2.1.4.2. Event interception, however, requires malware to divert the *control flow* of the infected system at runtime. To achieve this, malware must install *hooks* in the system that facilitate the required control flow transfer on behalf of the malware whenever the desired event occurs. Instead of detecting the malware itself, hook detection aims to find these hooks within the system to identify malware infections. In the following, we will first discuss the topic of malware and hooking in more detail, before we present an overview of existing hooking mechanisms.

Malware and Hooking. Petroni and Hicks [116] estimated that about 96% of all rootkits require *hooks* within the system to function. Intuitively, this makes sense: since the sole purpose of rootkits is to provide stealth, they have to hide all signs of an infection. While existing structures can be hidden using techniques such as *direct kernel object manipulation (DKOM)* [152], hooks enable rootkits to react to changes occurring at *runtime*. Consider, for instance, that a hidden process creates a new network connection or a child process. Naturally, a rootkit must also hide such newly created objects to achieve its goal. This, however, requires a rootkit to be notified of the occurrence of such events. Hooks solve this problem by enabling a rootkit to install callback functions in the system. This makes them an integral part of rootkit functionality.

In practice, rootkit functionality is often mixed with a variety of malicious payloads. According to a report by Microsoft released in 2012 [52], "some of the most prevalent malware families today consistently use rootkit functionality". The primary reason for this is that the single purpose of a rootkit is to avoid detection. Consequently, it is not a big surprise that the techniques formerly only found in rootkits are increasingly being adapted by malware. Since rootkits require hooks to function, this, however, also implies that any malware based on rootkit functionality will require the same. Hooks thus present an apparent place for malware detection.

Existing Hooking Strategies and their Detection. In general, we can distinguish between two different types of hooks: *code* hooks and *data* hooks [174, 184]. Code hooks work by directly patching the application's code regions: wherever the attacker wants to redirect the control flow of the application, she overwrites existing instructions with a branch instruction. As a result, the control flow of the application is diverted every time the execution passes through the modified instructions. Since code hooks modify existing instructions, however, they can be detected using code-based integrity checking as has been described in the last section.

Instead of modifying code directly, data hooks target *persistent* control data (i.e. function pointers) within the application. By modifying control data, the attacker is able to divert every control transfer that makes use of the modified data. For example, the most straightforward method for intercepting the execution of system calls is to modify function pointers within the system call table.

To counter the threat of data hooks, researchers proposed various systems that attempt to protect function pointers within an application (e.g [19, 81, 116, 174]). As in the case of data-integrity checking, most of the existing approaches thereby focus of the protection of function pointers within the kernel. In general, this is accomplished by ensuring that every function pointer points to a valid function according to the kernel's CFG. Petroni and Hicks [116] leverage such an approach, for example. In particular, they implemented a monitor that periodically validates all function pointers within the kernel's memory region. To obtain all function pointers, the monitor traverses the graph of all kernel objects by starting from the global variables and repeatedly following all pointers contained in the objects that it reaches. In the process, the monitor also extracts all function pointers contained within the objects it discovers along the way and verifies that each of them points to valid function. Once a validation is complete, the monitor starts the whole process anew.

2.5 Summary

This chapter laid the foundation that we require to address the research questions of this thesis. To determine whether data-only malware represents an equally realistic and powerful threat as traditional malware, we have to compare both malware forms. For this purpose, provided an overview of traditional malware and identified its key properties. In the process, we established that the dependencies of a malware form could be the key for its detection.

Next, we considered the second cornerstone for our in-depth analysis of data-only malware in Chapter 3: code reuse techniques. The main idea behind these approaches is to perform computations by combining existing instructions into a new program. The execution of this program is thereby controlled by a data structure which contains pointers to the instruction sequences that should be reused. By exploiting specific hardware or software features, the individual sequences can be connected in such a way that each sequence initiates the execution of the next. While this approach enables an attacker to perform complex computations without introducing code, the computational ability of code reuse ultimately depends on the instructions that the target application inherently provides. Consequently, code reuse techniques are another important aspect that must be considered to fully answer the question raised above. In addition, the hardware or software feature that a technique exploits in order to perform computations can simultaneously provide a basis for detection mechanisms, as we will see in Chapter 7.

From code reuse, we then moved to virtualization. In particular, we introduced VMI, which, due to its strong security guarantees, would provide an ideal platform for malware defense, if it were not for the semantic gap problem. Having detailed the capabilities of the technique, we thus discussed three approaches (in-band delivery, out-of-band delivery, and derivation) that can be used to bridge the semantic gap as well as their advantages and disadvantages. We will employ this information to create a VMI-based framework for malware detection and removal in Chapter 6.

Lastly, we provided an overview of important malware protection and detection mechanisms, which will allow us to analyze the effectiveness of existing defenses against data-only malware in Chapter 4.

With this information in mind, we can now address our first research question and determine the capabilities and limitations of data-only malware.

Chapter 3

Data-only Malware

The key idea behind data-only malware is to perform computations by combining existing instructions into a new malicious program. This is achieved by applying code reuse techniques to the problem of malware creation. While this approach enables the malware form to evade all code-based detection approaches, the question arises whether data-only malware constitutes as equally realistic and powerful a threat as its predecessor, given that it relies on code reuse to function. To answer this question, we will in this chapter determine the capabilities and limitations of data-only malware. For this purpose, we will perform a detailed analysis of the malware form based on the properties we identified in Section 2.1. In the process, we show that data-only malware is not only able to achieve all of the key properties of traditional malware, but even surpasses its traditional counterpart in terms of its level of stealth and its ability to evade detection. To prove the practicability of the approach, we will additionally provide detailed proof of concept (POC) implementations of sophisticated data-only malware that are capable of infecting current systems in spite of the numerous protection mechanisms that they employ. In our analysis, we will thereby discuss many previously unconsidered aspects of data-only malware, which are essential for the understanding of this malware form, but even more so for the development of countermeasures against it.

Chapter Outline. We begin by providing a definition for data-only malware, discussing the core challenges of its creation, and explaining the infection mechanisms that it can leverage in Section 3.1. Having discussed the fundamental principles behind the creation of data-only malware, we cover the three types of data-only malware that exist starting with the most simplistic type, "one shot data-only malware", in Section 3.2. While powerful in theory, this malware type faces the crucial constraint that it can neither react to events nor infect a system permanently. This finding will lead us to the discussion of persistent data-only malware and resident data-only malware in Section 3.3 and Section 3.4 respectively, which overcome these shortcomings. In Section 3.5 we then move from theory to practice and prove the validity of our considerations by providing

POC implementations of sophisticated data-only malware. Having discussed data-only malware in theory and practice, we conclude our analysis by comparing data-only malware with traditional malware in Section 3.6. In the process, we discuss the computational abilities of data-only malware, its level of stealth, its environment dependencies, and the applicability of signature evasion techniques such as encryption, polymorphism, and metamorphism. Finally, we summarize the chapter in Section 3.8.

3.1 Fundamentals

In this section, we cover the fundamental principles behind data-only malware. Before we can go into details on the creation of data-only malware, however, we first have to provide a definition for the malware form.

3.1.1 Definition & Differentiation

Data-only malware can essentially be seen as a malicious data-only program written in a code reuse language. The key property that thereby distinguishes data-only programs from traditional programs is that the former do not modify the *codebase* of a system. Consequently, to be able to provide a definition for data-only malware, we first have to specify what we actually mean by the term *codebase*:

Definition (Codebase)**.** *The codebase of a system consists of all intended instruction sequences residing within volatile or non-volatile storage that can be executed by the underlying hardware.*

Note that this implies that the codebase of a system is not fixed, but may change over time (e.g. when a new program is downloaded from the Internet). Following this definition, we will from here on refer to any data that is part of the codebase as "code", while we refer to all remaining data as "data". Based on our notion of codebase, we can now provide a definition for data-only malware:

Definition (Data-only Malware)**.** *Data-only malware is a program specifically designed to disrupt or damage a computer system without changing or extending its codebase.*

As the astute reader may have noticed, this definition of data-only malware is an extension of the definition of malware, which is commonly referred to as "software which is specifically designed to disrupt or damage a computer system" [123]. It contains three crucial aspects about data-only malware. First of all, data-only malware does not change or extend the codebase of a system. To accomplish this, data-only malware must *solely* consist of *data*. A crucial property of this class of malware is therefore that the IP never points to anything introduced by the malware itself.

Second, data-only malware is a *program*. That is, data-only malware consists of instructions for some form of machine, which enables the malware to perform computations [124].

This computational ability is what distinguishes data-only malware from plain malicious data. For example, consider data that purposefully crashes an application because it overflows a buffer and overwrites critical data of the application. Clearly, this data disrupts the normal operation of the system, but to qualify as data-only malware it must also provide the capability to perform computations without changing or extending the codebase.

Data-only malware achieves this by reusing the instructions of other applications. This leads to the third important observation about data-only malware: the malware form cannot exist on its own. Instead, data-only malware always requires a *host* application, whose instructions it can leverage, to function.

3.1.2 Core Implementation Challenges

Data-only malware is a data-only program. As in the case of a traditional program, the lifecycle of a data-only program can be divided into three stages: the *loading* stage, where the program is loaded into memory, the *execution* stage, where the program is executed, and the *cleanup* stage, where the leveraged resources are freed after the program terminated. However, while the OS sets up and supervises each of these stages for a traditional program, the realization and management of these stages must be conducted by the data-only program itself in the case of data-only malware. Whenever we want to implement data-only malware (or any other data-only program for that matter), we thus must find a way to realize each of these stages. In the following, we will describe each of the stages and the challenges it faces in more detail.

Challenge A: Loading and Storing the Data-only Program. Before a program can be executed, its code region must be loaded into memory. The equivalent of a code region in the case of data-only malware is the control structure. Consequently, one major challenge of data-only malware is to *load* this control structure and any other data required by the malware into memory and to *safely store* it during its execution. This requires that the infected application provides some functionality to move external data into memory. Hereby, it is essential that the memory that is used by the routine to store the data is large enough to contain the necessary control structure. In general, the control structure requires significantly more space than traditional shellcode.

Challenge B: Starting the Data-only Program. Besides loading the data-only program, it must also be *started*. That is, the virtual IP must be set to the control structure and its execution must be triggered. This step is comparable with setting the IP to the code region of traditional malware and essentially is what starts the actual execution of the data-only malware. How this is accomplished heavily depends on the location where the control structure resides (Challenge A) and the code reuse mechanism that is used. In the case of ROP, for example, the SP must first be set to point to the control structure in memory and the control structure must then be activated using a `ret` instruction. Finding a instruction sequence that achieves this is often a difficult task.

Challenge C: Restoring a Valid Execution Path. Data-only programs require the instructions of a *host* application to function. At some point during the execution of the host, the data-only program must be loaded into memory (Challenge A) and the virtual IP must be set to the entry point of the control structure (Challenge B). Achieving this requires the attacker to modify the state and the control flow of the host. For instance, to execute a ROP chain we must place the chain on the stack and execute a `ret` instruction. After the execution of the data-only program, we must restore a valid state and a valid control flow of the host to avoid side effects such as crashing the host application. Consider a data-only program that is loaded using a vulnerability in kernel space, for example. If the program does not restore a valid kernel control flow path after its execution, there is a good chance that the system will crash. Consequently, when designing data-only programs, one must not only consider the loading process of the control structure, but also the cleanup stage.

While we face a similar problem in the case of traditional malware (e.g. a virus that modifies its host in order to run), we can leverage arbitrary instructions to backup and restore any part of the hosts state in this case. For instance, we can easily preserve registers by pushing them on the stack and restoring them at the end of the execution. In the case of code reuse techniques, however, we have to find a gadget for each action that we want to perform, which makes this task naturally a lot more difficult as we can only work with the instructions that the host provides.

3.1.3 Infection Mechanism

One of the first questions that arises when we consider data-only malware is which attack vectors the malware can use to infect a system. Similar to traditional malware, data-only malware can make use of a vulnerability-based infection approach and a file-based infection approach. In the following, we will consider both infection mechanisms in more detail.

3.1.3.1 Vulnerability-based Infection

As code reuse techniques originally stem from the field of exploitation, vulnerabilities represent an obvious attack vector for data-only malware to infect a system. In fact, the techniques used by exploits can often be directly applied to load (Challenge A) and execute (Challenge B) data-only malware. The main difference arises in the following stages, where exploits in general make use of code, while data-only malware solely relies on code reuse.

How a vulnerability-based infection process is conducted in practice, heavily depends on the vulnerability that is exploited. Nowadays exploits are often quite complex, consist of multiple stages, and may require the combination of multiple individual vulnerabilities (e.g. control flow modification and privilege escalation) to succeed. We will provide a concrete example of a vulnerability-based infection process in Section 3.5.3.

3.1.3.2 File-based Infection

The second infection process that we want to consider within this thesis is *file-based infection*. Since there has, to the best of our knowledge, not been presented a file-based infection approach for data-only malware so far, we will consider this attack vector and its challenges in more detail. For the sake of simplicity, we will thereby focus on a file-based infection mechanism that employs an *executable* to infiltrate the system.

While file-based infection is in principle a very simple infection mechanism, realizing it solely based on data is a lot more difficult than in the case of traditional code-based malware. The main problem thereby lies in the loading (Challenge A) and activation (Challenge B) of the control structure. To see this we have to take a closer look at the loading process of an executable.

When a executable is started, it is first processed by a special program referred to as the *loader*. The purpose of the loader is to conduct all steps that are necessary to prepare the executable for its execution. As we discussed in Section 3.1.2, before a program can be executed its code and data regions must be placed into memory. In the first step, the loader thus allocates memory regions for the code and data regions of the executable and loads them into memory. Once this has been accomplished, the loader will update any address within the executable that depends on the a base address of another section. Consider a global variable, for example, that is accessed by a function of the program. Since the virtual address of this global variable depends on the base address of the data section, which is placed into memory by the loader, it is unknown at compile time. As a consequence, the loader must patch the correct address of the variable into the code section, once the code and data sections have been transferred into memory and the final location of the global variable is known. This process is referred to as *relocation*.

Besides using internal symbols such as the global variable from the last example, a program may also use external symbols such as functions from an external library as the libc. Therefore after relocation, the loader must also resolve any external symbols that the executable requires to function. This task is referred to as *symbol resolution* and essentially requires the loader to load any file into the virtual address space of the executable that is used by it. Finally, after *relocation* and *symbol resolution* have been completed, the loader can invoke the entry point of the executable in memory to start the execution of the actual program, which completes the loading process.

In the case of code-based malware, an attacker could thus simply modify the code section of the executable she wants to infect. As a result, the loader would load the manipulated code into memory and execute it similar to a benign program. Loading (Challenge A) and executing (Challenge B) the malware is straightforward.

Now consider that we want to accomplish the same using data-only malware. While we can also modify the data section of an executable to contain the data-only program, this will only place the malware into memory (Challenge A), it will, however, not trigger its execution (Challenge B). To achieve the latter, we must find a code sequence within the executable that activates the control structure. In the case of ROP, for instance, this implies that we must find a stack pivot sequence that sets the SP to our control structure and executes a `ret` instruction. Whether such a stack pivoting sequence exists, depends on the executable though. Additionally, even if such a sequence would be contained in the executable, how can we actually use it, if we do not know the address of the code region beforehand? Since the code region is loaded into memory by the loader, the final location of it is unknown *until* the loading process occurred, especially if protections such as ASLR are used. However, to employ a file-base infection approach we must attach the data-only program to a file *before* it is loaded leading to a "chicken and egg" problem.

Taking these problems into consideration, it becomes quite clear why there has not been presented a file-based infection approach for data-only malware so far. There is, however, an elegant solution: we can leverage the loader to load and start our data-only program at *runtime*. While the specifics of this approach depend on the file format that is used (e.g. ELF), the general idea behind this approach is as follows: as previously described the loader must perform relocation and symbol resolution to prepare a program for its execution. To conduct these steps the loader makes uses of management structures within the executable that specify what locations within the executable must be updated in what way. Since the loader can resolve any internal and external symbol, we can abuse it to construct the data-only program. To accomplish this we will manipulate the management lists used by the loader in such a way that it will first resolve the address of the gadgets we require and then write them to a predefined memory area. This will effectively construct our data-only program at a specified memory location at runtime. We will present a concrete example of such a file-based infection mechanism for the Linux ELF file format in Section 3.5.2.

While in principle any program that is involved in the execution of a binary can be abused for a file-based infection process, abusing the loader for this purpose leads to multiple advantages. First of all, since the loader is able to resolve arbitrary addresses, we can leverage any external symbol we want. This allows us to not only use gadgets from the infected executable, but from any library of the system. Second, a file-based infection process based on the loader will not just work for a single executable, but potentially for every executable it loads. Consequently, the proposed approach is not only powerful, but also widely applicable.

3.1.4 Prerequisites

Based on the observations we made throughout this section, we can state the *prerequisites* that must be fulfilled for the successful infection and execution of data-only malware. First of all, not every executable or vulnerability is suited for the execution of data-only malware. Instead, there are specific constraints that must be fulfilled. Most importantly, the executable or the vulnerability must allow the attacker to control the IP. This is important as the attacker cannot introduce code, yet needs to modify the control flow. In general, this is done by careful manipulation of the IP. For example, in ROP this is achieved by constructing a control structure with pointers to gadgets. If the control structure is properly constructed, the functionality of the `ret` instruction can be leveraged to control the execution flow. However, before this structure can be used, an attacker must find a way to set the SP to its location. For the general case of data-only malware, this therefore implies that the infection process must not only provide control of the IP, but must also enable the attacker to activate her control structure (Challenge B).

In addition, the infection mechanism must provide the attacker with the ability to transfer the required control structure into memory (Challenge A). This requires that the host application provides some functionality to move attacker controlled data into memory. The memory area that is controlled by the attacker must thereby be large enough to contain the control structure.

Lastly, the host program must provide the instruction sequences (i. e., gadgets) that are necessary to implement the functionality of the data-only malware. This leads to the following prerequisites for data-only malware:

Instructions The victim's system must contain the instruction sequences that are required to implement the malware's functionality. In general, this implies that the data-only malware requires a specific host application to be running on the victim's system whose instructions it can leverage to implement its own functionality.

Infection The victim's system must provide an infection point: either a file that can be manipulated to start the execution of the malware or a vulnerability.

Memory The infection mechanism must provide a mechanism to load the required control structure into memory.

Control The infection mechanism must provide the attacker with control of the instruction pointer (IP) and enable her to activate the control structure.

3.2 One Shot Data-only Malware

Having discussed the fundamental principles behind data-only malware and the infection mechanisms it can leverage, the question arises which in-memory strategies are available to this malware form and whether it can achieve persistence and residence (see Section 2.1.4.2 and Section 2.1.4.3 respectively). To discuss these properties, we will throughout the next three sections consider three basic types of data-only malware that each exhibit one of the properties: one shot data-only malware, persistent data-only malware, and resident data-only malware. We begin with the simplest type: one shot data-only malware.

In its simplest form data-only malware is neither persistent nor resident, but instead solely leverages a direct-action approach. In this case the malware essentially consists of a single data-only payload that is executed once, performs the desired action, and exits. This is why we refer to this malware form as "one shot" (data-only) malware. While one shot malware does not have a very sophisticated design, it is the data-only malware form that has been studied most commonly in the limited research available.

Since one shot data-only malware is not persistent, it does not permanently change the control flow of a system. Instead, actions are performed by executing an infected executable or exploiting a vulnerability over and over again. At its heart, one shot data-only malware thus consists of a loading stage that can handle different data-only programs. Each action that the malware supports is implemented as an individual program. To execute a particular action, the loading stage is triggered using the corresponding program. While one shot malware can thus provide multiple different functionalities in the form of different programs, the execution of a particular functionality requires the reexecution of the entire malware. Hereby, the execution of the malware must always be triggered by an external entity. Since one shot malware is neither persistent nor resident, it can *never* trigger its own execution.

While one shot data-only malware can in theory use a file-based infection approach for the loading stage, the one shot malware that has been presented so far exclusively uses a vulnerability-based infection approach. Certainly, one of the main reasons for this is that implementing one shot malware using a vulnerability is straightforward. Essentially all that an attacker requires is an exploit that can handle arbitrary data-only payloads. Modifying an existing exploit to support this feature is generally easy. To show this, let us reconsider the core problems of data-only malware that we identified in Section 3.1.2 and analyze how they can be solved in the context of one shot data-only malware. In the process we assume that we already have a control flow modifying exploit at our disposal that we want to use as a basis for our attack.

Challenge A: Loading and Storing the Data-only Program. The exploitation of a control flow modifying vulnerability requires the attacker to have the ability to load a control structure or shellcode into memory. After all, she must be able to execute her own code. Therefore the attacker can simply reuse the loading mechanism of the exploit for the implementation of the one shot data-only malware. The only problem that might occur is that the control structure of data-only malware usually requires more space than an optimized shellcode. However, this issue can be solved with the help of a multi-staged loading process. In this case, the attacker makes use of a small and optimized control structure which will upon execution load a larger control structure.

Challenge B: Starting the Data-only Program. Since most applications nowadays are protected by mechanisms such as W $\oplus$ X, exploits in general also have to use code reuse mechanisms during the first stage of the exploitation. Consequently, the attacker will in many cases be able to simply reuse the activation mechanism of the exploit for the activation of the one shot data-only malware. Otherwise, if the exploit is capable of directly executing code, there is from a practical point of view no need for a loading process that is based on code reuse techniques in the first place. Instead, the attacker could simply use code to load the one shot malware into memory and to start its execution.

Challenge C: Restoring a Valid Execution Path. Once the payload has been executed, an exploit must in general restore a valid execution path. While it is possible to simply let the exploited application crash (given that it is not critical for the health of the system), this may lead to suspicious entries in the log files of the attacked system, which should be avoided. If the assumed exploit restores a valid execution path, this mechanism can directly be reused by the one shot data-only malware similarly to the aforementioned challenges. Otherwise the attacker has to find a way to restore a valid execution path and add it to the end of the payload. A well-known mechanism that is often used to achieve this in the case of exploitation is to restore the *original* execution path. To do this, the attacker simply returns to the original intended location within the application that would have been invoked if the vulnerability had not been exploited.

Conclusion. The core challenges of one shot data-only malware which uses vulnerability-based loading stage are very similar to the challenges that we face in traditional exploitation. In fact, vulnerability-based one shot data-only malware can be seen as an exploit with a sophisticated payload. This is why it is often simple to leverage existing exploits for the creation of one shot data-only malware. One shot data-only malware is a natural extension to traditional exploitation. Persistent and resident data-only malware is, however, much more difficult to realize as it significantly differs from the problems we encounter in traditional exploitation. As does a file-based infection approach.

The reason that data-only malware has mostly been non-persistent so far is that it is very difficult to achieve persistence without introducing any code. In the next section we will take a closer look at the challenges involved and how one might overcome them.

3.3 Persistent Data-only Malware

As described in Section 2.1.4.2, persistence is a crucial property that many malware forms require. Upon initial consideration, it seems unlikely that persistence is possible without the introduction of code. In fact, some researchers dismiss the possibility of persistent data-only malware entirely [128], while other researchers have speculated that it is possible, but did not manage to actually implement it [24]. In this section, we will show that persistent data-only malware is, indeed, possible and will provide a concrete architecture for its implementation.

In contrast to one shot data-only malware, persistent data-only malware is capable of permanently altering the normal control flow of a software system. Loading the malware is achieved by executing a manipulated binary (file-based infection) or by exploiting a vulnerable program (vulnerability-based infection) once. This infection process has essentially two stages. The first stage is the *initialization stage*. During this stage the *initialization control structure* is executed, which performs the bootstrapping of the malware. In particular, the initialization control structure is responsible for placing the hooks, which will later trigger the execution of the malware, and the loading of the *persistent stage*. This persistent stage then implements the persistent functionality of the malware. While this sounds very straightforward, there are several challenges that one must overcome in order to be successful. These are described in the following section.

3.3.1 Challenges

Persistent data-only malware faces four fundamental challenges that are directly related to the three core implementation challenges that we identified in Section 3.1.2. However, before we describe these challenges in more detail, it is important to separate the challenges faced in the initialization stage and the challenges faced in the persistent stage. As it turns out, the initialization stage must face the same challenges that one shot data-malware faces. The reason for this is that the initialization control structures essentially performs all the tasks that are necessary to prepare the execution of the persistent stage. For this purpose it is executed once during the initialization stage of the persistent data-only malware. It is therefore comparable to a payload of one shot data-only malware. Since the challenges of one shot data-only malware have already been detailed in the previous section, we will in this section only consider the challenges that the persistent stage faces. That is, the challenges that have to be overcome before, during, and after the execution of the persistent stage. Consequently, in the following, we will assume that the initialization control structure of the malware has already taken control of the system and now prepares the execution of the persistent stage.

3.3.1.1 Finding a Suitable Memory Location

First, a memory location must be located that can contain the persistent control structure of the malware (Challenge A). In contrast to the case of non-persistent data-only malware, it is essential that this memory location is exclusively owned by the malware itself in order to avoid the control structure being destroyed during the normal execution of the infected program. The problem arises since the persistent control structure is not just executed once as in the case of one shot data-only malware, but every time a hook is invoked. As a result the stack is usually not suited for such a task. Instead, a memory area must either be reserved within the system or an existing unused memory area can be occupied. The latter is, for instance, possible if the infected application does not make full use of a data region that has been allocated to it. Finally, care must also be taken that this memory location is never deallocated after the initial stage has taken place.

3.3.1.2 Protecting Against Overwrites

Second, the persistent control structure has to be protected against overwrites (Challenge A). If the control structure is modified in an uncontrolled way, it is very likely that the malware will malfunction on the next execution. Notice that finding a memory location that is exclusively owned by the malware as described in the previous challenge, is not enough to guarantee that the persistent control structure is not overwritten. In the case of ROP, for instance, we have to set the SP to point to the persistent control structure to execute it. If another thread of execution interrupts our control flow and tries to make use of the stack before we finish, it could overwrite gadgets of the persistent chain that have been executed before we were interrupted.

In general, there exist two possible types of overwrites: self-induced and interrupt-induced. The former refers to overwrites that are triggered by the malware itself. As an example, consider a `call` instruction that is part of a gadget used within a ROP chain. This instruction will essentially push the return address on the stack and then transfer control to the location specified by its operand. Since the SP points to the control structure, the call instruction will overwrite parts of it by pushing the address. In fact, the push will, in many cases, overwrite the address of the gadget that contains the `call` as shown in Figure 3.1. This is due to the fact that the address of the gadget that is currently executing (A in Figure 3.1) usually resides directly before the current SP.

While self-induced overwrites have to be kept in mind when designing persistent data-only malware, they can be avoided by carefully selecting the gadgets that are used to implement its functionality. Interrupt-induced overwrites on the other hand, are overwrites that are triggered by an external event and can therefore not simply be avoided. Instead, the malware must be designed to protect itself against these overwrites. Due to the fact that interrupts are very frequent events, it is very likely that persistent kernel malware is interrupted during its execution and an interrupt handler is invoked. This interrupt handler may, amongst other things, make use of the current stack during its execution. In the case of ROP this means that the part of the control structure that resides before the current SP may be overwritten.

The types of overwrites that can occur heavily depend on the technique that is used to to implement the malware (e. g., ROP) and its functionality. Interrupt-induced overwrites, for instance, usually only occur within kernel space, but not in user space.

3.3.1.3 Resuming the Original Control Flow

Third, since the persistent stage of the malware is invoked by a function hook, we have to make sure execution continues normally after the malware has run (Challenge C). This is due to the fact that the execution path, which led to the invocation of the hook, will most likely expect a result from the original function that was replaced by the malware. This result has to be provided by the malware. Otherwise, if the malware would simply try to gracefully terminate the execution path (e. g., by returning to the main function), certain

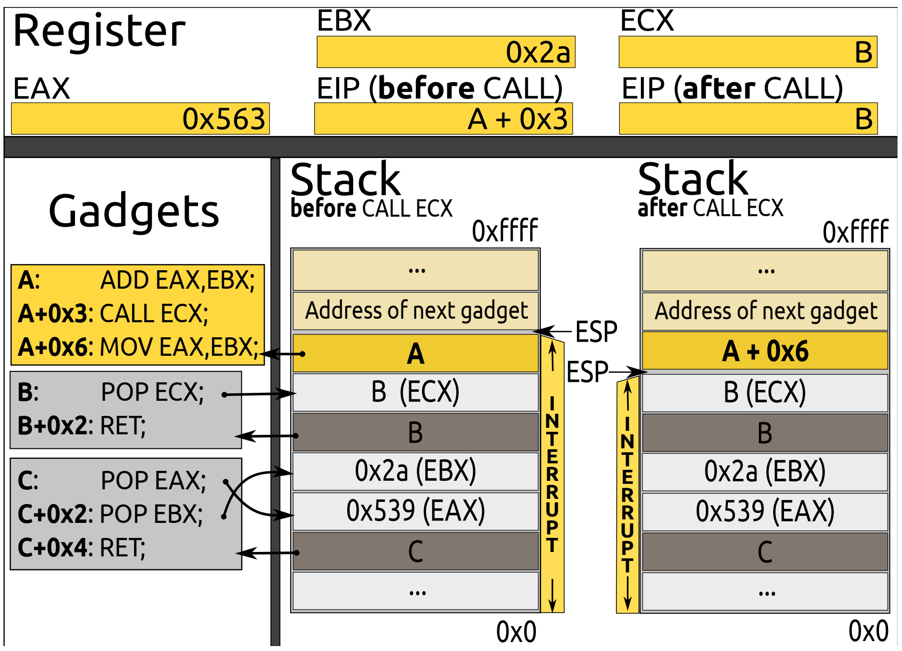


Figure 3.1: Self-induced and interrupt-induced overwrites in the case of ROP. To visualize self-induced overwrites, the picture shows the state of the machine before and after the execution of the `CALL ECX` instruction at address A + 0x3. Interrupt-induced overwrites are displayed using the bars next to the stack, since they could overwrite any value before the current SP. The ROP chain that is shown uses three different gadgets to load two immediates into EAX and EBX, add their values, and store the result in EAX.

code paths would never be executed, which will lead to a reduction in functionality and in the worst case a system crash. This is especially a problem for kernel space persistent data-only malware, where a failure to restore a valid execution path, will in most cases crash the entire system.

To be able to continue the original execution path, persistent data-only malware must be careful to not overwrite register or memory values during its execution that it might need later on to resume the execution path. This essentially requires that the persistent stage backs up important registers/memory locations *before* it makes use of them, unless the malware can predict/infer their values. Additionally, the malware must restore the original values before it hands back the execution.

3.3.1.4 Activating the Control Structure

Fourth and more importantly, a mechanism or specific instruction sequence must be found that activates the persistent control structure when a hook is invoked (Challenge B). If we once again consider ROP as an example, it is not enough that the IP is manipulated through an overwritten function pointer, but we must also manipulate the system in a way such that the SP points to the beginning of our persistent ROP chain. Since this chain is stored somewhere in memory (not on the stack) the first instruction sequence that is executed on behalf of the malware when the function hook is invoked must modify the SP to point to the ROP chain. That is, it must *switch the stack*. This *switching sequence* is a requirement for a persistent data-only malware. Notice that the switching sequence must in general be a sequence of continuous instructions. As we do not yet have control over the SP at this stage it is very difficult to build a chain of multiple gadgets. Setting the SP to a specific value under these conditions is quite challenging.

At first glance, it seems as if the previously described challenge is the same challenge that we face in a data-only exploit. In this case we also need to find a way to activate our control structure, which for ROP means that we have to point the SP to our volatile stack as described in Section 2.2.1.2. However, while the problems are related, the scenarios in which we try to solve them are very different. To see this, lets us briefly compare the machine state in both situations.

In the case of a traditional ROP exploit, we usually have a very solid understanding of the state that the machine will be in when our exploit is triggered. For instance, consider an exploit that overwrote the saved IP on the stack. When the vulnerable program tries to return to this address, attacker controlled code will be executed. Since we overwrote the return address of a particular function, we know which functions will be executed before the overwritten return address is used. Due to the deterministic nature of these functions, we are able to gather a lot of information about the machine state. In particular, we will know the layout of the stack and the data types of the general purpose registers and the local and global variables, which will allow us to use techniques such as “ret2reg”. In the case of a traditional stack-based overflow, we can even simply place our ROP chain on the stack by writing past the saved return address.

In contrast to this, consider persistent data-only malware which is invoked by a hook placed somewhere within the system. Depending on the location of the hook, there might be dozens of execution paths that lead to the invocation of the hook. In general, we will therefore not be able to make any assumptions about the stack or the general purpose registers. In short, the only register whose value we can predict when the hook is invoked is the IP. Since we only control the IP, but have not yet activated any control structure, we will only be able to use a single gadget to switch the stack. This is the worst case scenario that we described in Section 2.2.1.2. However, in this case we neither control another register nor a buffer on the stack. Consequently, we cannot simply use common stack pivot gadgets such as the one presented in Section 2.2.1.2. Activating our persistent control structure in this situation is a difficult problem. What is even more, we will need to find a way to conduct the stack switch without corrupting register or memory values that are needed later on. After all we must hand back the execution to the previous function after we have handled the hook in order to avoid any side effects on the system as has been described in the previous section. This is also not the case for traditional exploits, where a graceful exit is in general enough to avoid a crash of the system.

An ideal initialization sequence for a ROP-based piece of malware on the x86 architecture might look as follows:

Listing 3.1: An ideal stack switching gadget.

```
mov eax, esp ; store the current ESP in EAX
mov esp, &control_structure ; move control structure address into ESP
ret ; trigger the control structure
```

However, it is obviously very unlikely that such an instruction sequence exists. In the following section, we discuss hardware and software-based solutions that can be used to manipulate the SP when a hooked function is called for a ROP-based approach.

3.3.2 Hardware Mechanisms

All of the following hardware-based mechanisms require the highest privilege-level to use them. Therefore these mechanisms are mainly of interest for attacks on the kernel.

3.3.2.1 The Sysenter Instruction

The `sysenter` instruction was introduced by Intel with the Pentium-II processor as a replacement for the interrupt-based system call mechanism. Since `sysenter` fulfills all the tasks that are required for a switch from a lower privilege level to the highest privilege level without intermediate table look-ups, it is much faster then the previously used interrupt-based system call invocation [64]. As a result, all modern OSs support the use of the `sysenter` instruction as a alternative to interrupt-based system calls.

Internally, `sysenter` relies on three model-specific registers(MSRs) to perform a context switch from a lower privilege level to ring 0. Namely these MSRs are:

IA32_SYSENTER_CS	Defines the target code segment that will be used after the context switch.
IA32_SYSENTER_EIP	Holds the IP that will be used after the context switch occurred.
IA32_SYSENTER_ESP	Holds the SP that will be used after the context switch.

By carefully manipulating the `IA32_SYSENTER_EIP` and the `IA32_SYSENTER_ESP` MSRs, an attacker can control both the SP as well as the IP. In order to leverage this approach to place hooks within the system, the malware would first need to set the appropriate MSRs to point to the malware's persistent control structure (SP) and the first gadget (IP). The hook itself then needs to point to a `sysenter` instruction within memory. As a result, every invocation of the hooked function would transfer the execution control to the malware.

A problem with such an approach is that the current SP is not saved anywhere and is simply overwritten. Therefore extra steps must be taken to restore the original SP after the malware executes. How this is achieved heavily depends on the particular hook and OS used.

Finally, it may seem that such an approach would break the original system call mechanism, however this need not be the case. First, it is often the case that 64-bit OSs prefer to use yet a third mechanism for implementing system calls, namely the `syscall` instruction. In this case, we need not worry as the `sysenter`-based mechanism is not in use anyway. On the other hand, if the host does make use of the `sysenter` instruction, the malware must simply handle this case and determine whether the call to our hook is the result of a "real" system call or the result of a function hook and react appropriately. In the case of a "real" system call, the malware simply needs to hand control to the system call dispatcher.

3.3.2.2 The Task State Segment

Although the feature is not used by most modern OSs, the x86 architecture provides a hardware mechanism for performing context switches between processes. For this purpose there exist so-called task-state segment (TSS) descriptors, which are part of the global descriptor table (GDT). By invoking a TSS descriptor, an attacker can load a completely new execution context. Consequently, persistent data-only malware can make use of this feature to control the IP as well as the SP during the invocation of a hook. To achieve this, the malware must first set up a TSS descriptor and then point the function hook to an instruction sequence that invokes this descriptor. A *far jump* to the descriptor is often used for this purpose[1] (`jmp <tss_desc>:0x0000`).

When the hook is executed the machine will then use the TSS descriptor to perform a "context switch" to the malware. During this process the hardware will first save the value of all general purpose registers in the TSS descriptor of the current task, before setting them to the values stored in the just activated TSS descriptor. This allows the malware not only to load a completely different execution context, but also enables it to easily access and restore the old execution context.

While this approach is very powerful, it is restricted to 32-bit systems. On 64-bit systems the x86 architecture no longer supports the above described context switching feature. However, while context switching has been disabled, a new mechanism has been introduced to the TSS that similarly allows an attacker to control the SP, the Interrupt Stack Table (IST). The IST is essentially a table of pointers, where each pointer contains the address of a memory region that can be used as stack region by an interrupt handler. This mechanism allows the kernel to individually assign a stack from the IST to each interrupt handler.

In the x86 architecture, interrupt handling is based on the IDT. The IDT contains an interrupt-gate descriptor for each individual interrupt. Amongst other things this descriptor specifies the address of the interrupt handler that should be invoked. Whenever

[1]The interested reader can find an overview of other possible sequences in Section 7.3 of the Intel Software Developer's Manual 3A [64].

an interrupt occurs, the number of the interrupt is used as an index into the IDT. Based on the number of the interrupt, it is therefore possible to obtain its corresponding interrupt-gate descriptor, which in turn specifies the address of the interrupt handler.

Besides the address of the interrupt handler, the interrupt-gate descriptor also contains an index into the IST. Should this index be greater than zero, the hardware will load the address contained within the specified IST entry into the SP, before invoking the interrupt handler. In addition, the machine will push the old value of the SP and the IP as part of the interrupt-stack-frame onto the new stack such that the interrupt handler is able to restore their original values after its execution.

To use this mechanism for a stack switch, the initialization stage of the malware has to point one of the entries within the IST to the location of the persistent stage. Additionally, one of the interrupt gate-descriptors in the IDT has to be setup to use this modified IST entry and to point to the first gadget in our persistent ROP chain. This first gadget must increase the SP by the size of the interrupt-stack-frame as the hardware automatically pushes this frame to the new stack when the interrupt is invoked. Finally, the hook has to be set to a gadget that invokes the interrupt whose descriptor was prepared in the way just described. The invocation of the hook will then lead to an interrupt, which will in turn lead to stack switch that in combination with the stack increasing gadget will lead to the execution of the persistent chain. Once the persistent chain finished its execution, it can restore the original SP and IP from the interrupt-stack-frame [64].

3.3.3 Software Mechanisms

Having described some hardware mechanisms in the previous section, we will now introduce several software mechanisms for switching the stack. These software-based mechanisms are specific to the hook that is placed within the system. Therefore, these approaches are not universal, but serve as examples of what is possible.

3.3.3.1 Using a fixed Memory Address

The easiest way to achieve a stack switch is to find a gadget that loads a *fixed* address into the SP and to store the required control structure at this fixed location. This enables us to perform the stack switch by simply invoking the gadget. Besides our own work, this is the only other known stack switching technique that we are aware of. It was first proposed by Hund [61].

While the approach is simple, it faces multiple problems. First of all, gadgets that load a fixed address into the SP are not very common. This is especially true for the x86-64 architecture. Consequently, this approach may not be available in practice. Second, even when one finds such a gadget, the memory area located at the fixed address may already be in use. In this case, we cannot simply place our control structure at the fixed address as we might overwrite important information. Third, given that the memory area is free, we need to find a way to actually load our control structure to this position. If we simply manipulate the page tables and place the structure there, we risk that our control structure is overwritten, should the memory area ever be allocated, as has been described in Section 3.3.1.1. Thus to provide a reliable solution, we must find a way to force the memory allocator to assign the memory area to us where the fixed address resides. While techniques such as heap spraying [31] may allow us to influence the allocator, obtaining a specific address can be difficult. In addition to allocating the address, the memory range available at the address must be large enough for our control structure. Taking all these problems into consideration, the technique seems to be only useful in particular scenarios and not as a general switching mechanism.

3.3.3.2 Adapting the Location of the Control Structure

The main problem that hinders us from using a common stack pivot gadget such as the one described in Section 2.2.1.2 to switch the stack is that we neither control a register value nor a buffer on the stack when a hook is invoked. However, we might be able to control the location in which our persistent ROP chain will reside. If this is possible, we can circumvent this problem by placing the persistent control structure of the malware above the stack of the process. By this we simply mean that the control structure must be loaded at an address that is smaller than the original stack base minus the maximum stack size of the process. A switch to the persistent control structure can then be performed using the common stack pivot `sub esp, <offset>; ret;`.

Since the malware control structure resides above the process stack, it will not be destroyed during the normal program execution given that the maximum stack size of the process is known. In addition, we are not restricted to a single fixed address, but any location near the stack will suffice. Whether it is possible to place the persistent chain at such a location depends on the structure of the vulnerable program and the employed infection mechanism. In general, an attacker can apply exploiting techniques such as heap spraying [31] to influence memory allocation. While the approach is still quite constrained, it is not as restrictive as placing the control structure at a fixed location.

Finally, notice that the constant stack offset must not necessarily point directly to the malware's control structure. Instead the attacker can introduce a NOP sled at the beginning of her control structure (in ROP terms a NOP is simply the address of a `ret` instruction). In this case the constant offset must simply point somewhere into the NOP sled. The success of the approach will then depend on the variation of the stack –which generally occupies only one or two pages– and the size of the sled.

3.3.3.3 Adapting the Location of the Stack

Instead of placing the persistent control structure at a suitable location above the stack, it is also possible to change the location of the stack itself. This is due to the fact that OSs usually store the address of the stack for each process and the kernel in specific registers or memory locations. In the case of Linux, for example, the kernel SP is stored within a `per_cpu` variable. When the kernel switches from user space to kernel space, it will load the address stored at this location into the SP. Thus by overwriting the stored address, an attacker is able to set the kernel stack to an arbitrary memory location. Similarly, an attacker can overwrite the saved SP of a single or multiple processes.

If the attacker controls the SP through this technique, she can, for instance, place the SP in front of the persistent control structure. The stack switch can then be performed using a gadget such as `add esp, <offset>; ret;`.

3.3.3.4 Using Pointer Chains

One of the biggest problems that persistent data-only malware faces when attempting to set hooks is that until the stack is switched (in the case of ROP) it can only rely on a single sequence of instructions to perform the necessary task of activating the control structure. If the malware could chain multiple instruction sequences this task would be much easier since it could combine multiple gadgets to reach this goal.

One possibility that would allow one to create a small chain of instruction sequences is to overwrite multiple function pointers that are called in sequence. To demonstrate this, consider the following example: The vulnerable program contains a global buffer that is located in the data section. Imagine that the initialization stage loaded the persistent control structure in this buffer. It is very likely that the vulnerable program contains various instruction sequences that operate on the buffer. For example, there may be a 'strncpy' operation that copies data into the buffer. On the x86 architecture this could result in the following assembler code:

Listing 3.2: Call to `strncpy` in assembler.

```
1  mov [esp + 8], size;
2  mov [esp + 4], &source;
3  ; move absolute address of the global buffer
4  ; to the top of the stack
5  mov [esp + 0], &dest;
6  call <strncpy@plt>
```

This provides the malware with an instruction sequence that loads the absolute address of the buffer onto the stack (Line 6). In addition, the malware can control the function call in Line 7. This is due to the fact that library functions (e. g., strncpy) are called as function pointers that are offsets within a global table. In Linux this table is called the global offset table (GOT) while in Windows this table is referred to as the Import Address Table (IAT). If we continue with our example, overwriting the function pointer of the strncpy function within the GOT[2] allows the malware to execute a second instruction sequence that loads the absolute address into the SP.

Note that while this is a contrived example, it is quite generic and the constructs used are very common practice. The prerequisites for such an approach are (1) the

[2]Notice that this overwrite has to occur during the initialization stage. When the persistent stage of the malware is invoked by a hook, the function pointers must already be overwritten.

existence of a global buffer, (2) a library function that operates on that buffer, and (3) a writable table that facilitates the linking of library functions. In both Windows and Linux environments these are commonly found in processes.

3.3.4 Architecture

Up to this point, we have presented a rather abstract view of the architecture of persistent data-only malware. To discuss the challenges associated with the creation of data-only malware and the mechanisms that can be used to activate it, we considered two stages: the initialization stage and the persistent stage. In this section we want to refine this view and present a concrete architecture for persistent data-only malware. This architecture is shown in Figure 3.2. As one can see, the architecture makes use of four different control structures: the initialization chain, the copy chain, the dispatcher chain, and the payload chain. While the initialization stage of the malware only consists of a single control structure (initialization chain), the persistent stage has been divided into the copy chain, the dispatcher chain, and the payload chain. In the following we will describe each of the chains in more detail. In the process we will also state which of the previously described challenges each individual chain faces.

3.3.4.1 Initialization Chain

As has been described in Section 3.3, data-only malware is loaded using a specially crafted binary or a vulnerability. The component that is executed during this initial loading phase is the the initialization chain (1). Since this component is only executed *once*, it acts very much like more traditional ROP exploits, which means that it does not require an exclusive memory area and is not affected by overwrites as outlined in Section 3.3.1.1 and Section 3.3.1.2, respectively. In addition, the initialization chain usually does not have to restore the original execution path as outlined in Section 3.3.1.3. Instead, any execution path that leads to a graceful exit is in general sufficient.

The initialization chain is responsible for conducting all steps that are necessary for bootstrapping the execution of the persistent stage. In particular, it must place a hook (2) within the victim's system, setup a switching mechanism (3) as outlined in Section 3.3.1.4, and copy the copy chain (4a) into memory. The latter requires that the initialization chain solves the first challenge discussed in Section 3.3.1.1. That is, the initialization chain must copy the copy chain to a memory location that is exclusively owned by the malware.

In addition, the initialization chain may have to create global state (4b), if the malware requires this. State is essential if the malware requires data to be stored across multiple invocations. Such a data area can either be integrated into the copy chain or be placed at a separate memory location as shown in Figure 3.2. In any case, the memory region used to contain the state must - similar to the copy chain - be exclusively owned by the malware.

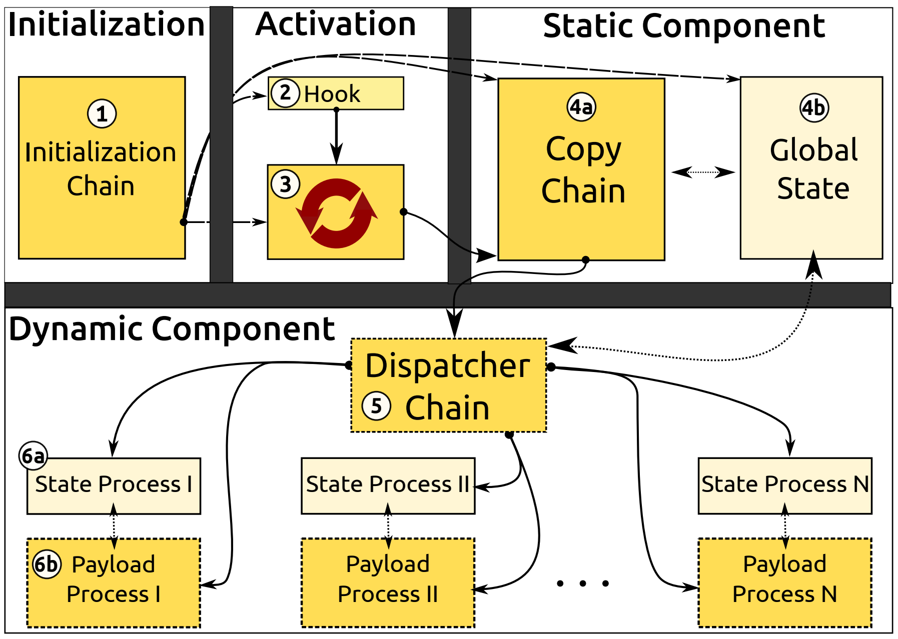


Figure 3.2: Overview of the proposed architecture for persistent data-only malware.

3.3.4.2 Copy Chain

The copy chain is invoked every time the hook that the initialization chain placed is triggered. In particular, the hook will transfer control to the switching mechanism, which in turn will invoke the copy chain.

The copy chain is the only truly persistent chain of the malware. Due to this fact it faces the most restrictions and must be carefully created to avoid overwrites as outlined in Section 3.3.1.2. It fulfills two main tasks. First and foremost, it must save the values of all general purpose registers when it begins execution in order to be able to restore the original register values after the malware has been executed as outlined in Section 3.3.1.3. To achieve this, the malware may only leverage gadgets that use registers which have already been saved, must not be saved according to the calling conventions, or whose values can be predicted. Consequently, this chain is severely limited when it starts execution, but will have access to an increasing number of gadgets with every register it saves. The values of the registers can be stored in the global state.

However, even when all registers have been saved, the copy chain is still tightly restricted as it must be executed with interrupts disabled and cannot invoke external functions to avoid overwrites as discussed in Section 3.3.1.2. Such restrictions could severely limit the functionality of the malware. To solve this problem, the copy chain creates a separate dynamic component upon each invocation of the hook. To do this, it simply copies the next control structure (the dispatcher chain in case of Figure 3.2) to a predefined memory area that is created by the initialization chain. Once activated, this dynamic control structure can then execute without having to consider self-induced overwrites as it is dynamically created on the fly for every invocation of the hook.

While this approach is sufficient for malware that only infects a single process, kernel-level malware may set a hook that can be triggered by multiple executions paths that are running concurrently. Consider a hook on a system call, for instance. When a system call is invoked, it is not guaranteed that the system call execution will return before a context switch and a different process makes the same system call. As a result, the same hook may be triggered multiple times by different processes simultaneously. In such a case, it is possible that the dynamic component of the previous system call hook is overwritten by the currently executing hook. This situation would likely lead to a kernel crash once the context and execution of the previous process is restored.

The most straightforward method to avoid this may simply be to disable interrupts. While we make use of this method in the copy chain, it is not an ideal solution for the entire dynamic component. This is due to the fact that this would lead to a further constraint for the malware. This constraint being that the malware may not use any external functions, since they may reenable interrupts during their execution. As we want to keep the malware as constraint-free as possible, we look to a more elegant solution. To this end, we make use of a dispatcher chain (5).

3.3.4.3 Dispatcher Chain

A dispatcher chain (5) is required whenever multiple concurrent threads of execution can invoke one of the hooks used by the malware. The general idea behind a dispatcher chain is to create an individual payload for each process. To achieve this the chain allocates an individual memory area for each process and copies the payload chain (6b) into this memory area on each invocation occurring in the context of the process. Similarly, the dispatcher will create individual state (6a) for each process, where information such as the register values for this process, which are copied from the global state, are stored. The dispatcher must therefore also guarantee that a specific payload chain will always have access to the same state area. To do this the dispatcher must patch the address of the state area into the payload chain at runtime.

This approach provides each process with individual persistent state and a unique payload and thus effectively avoids the problem described above. Finally, notice that the dispatcher chain must be directly invoked by the copy chain and can only make use of external functions that do *not* enable interrupts. In other words, interrupts must remain disabled while the dispatcher chain is executing.

3.3.4.4 Payload Chain

The payload chain (6b) contains the actual functionality of the malware. Since it is recreated on each invocation by the dispatcher chain and is additionally unique for each process, it is neither affected by self-induced overwrites nor by interrupt-induced overwrites. That is, the malware can, at this point, invoke any external function and can make use of any register that has been saved by the copy chain. Thus the payload chain is essentially a traditional ROP-chain with the benefit that it may make use of persistent state to store data between invocations. As a result, the payload chain is very flexible and is only limited by the gadgets that the victim's system provides.

At the end of its execution, the payload chain must restore the original register values and hand control back to the execution path that was executed before the hook was invoked as outlined in Section 3.3.1.3. While the former can be easily achieved, since the register values are saved by the copy chain and placed into the process state by the dispatcher chain, the latter requires the restoration of the original SP. Since this information may be lost (depending on the switching mechanism that is used), this process is usually application dependent and must be solved on a case by case basis. In general, the FP can be used to restore the original SP given that the frame size of the function executing before the hook is invoked is known.

3.3.5 Summary

In this section we explained how data-only malware can achieve the important property of persistence. First, we must find a memory area that is exclusively owned by the data-only malware. While this can simply be accomplished by reserving new memory for the data-only malware, this alone is insufficient to protect persistent data-only malware from overwrites. Instead, we must also handle self-induced and interrupt-induced overwrites, which are the second challenge of persistent data-only malware. To cope with these problems we have to carefully design our persistent chain and disable interrupts at the beginning of its execution. This led us to third and most important challenge of this malware form, how we can force the activation of the persistent chain when a hook is invoked. To address this issue, we presented multiple hardware-based and software-based stack switching mechanisms that can be leveraged to achieve this task. In this context we also discussed the final challenge of persistent data-only malware: the restoration of the original control flow after the execution of the hook. Finally, we proposed an architecture for the creation of data-only malware that considers all of the aspects that we covered throughout this section. Most importantly, the architecture leverages an individual payload chain for each process, which enables persistent data-only malware to prevail in the otherwise devastating scenario where the stack is used by the system and the malware at the same time. We will present a POC implementation that leverages this architecture in Section 3.5.3.

3.4 Resident Data-Only Malware

While persistent data-only malware is, due to its ability to react to events within the system, far more powerful than one shot data-only malware, it still faces a severe restriction: it is unable to survive a reboot without human interaction. In general, however, malware aims to infect a system permanently and not just until the occurrence of the next reboot. To achieve this, the malware must become *resident*. In this section, we will cover the property of residence in more detail.

In order to survive a reboot, data-only malware must be automatically loaded (Challenge A) and executed (Challenge B) on system boot. How this can be accomplished, depends on the infection mechanism that the data-only malware uses. In case of a file-based infection mechanism, achieving residence is actually quite straightforward. Since the approach enables data-only malware to *permanently* infect an executable, the malware will become resident if the infected file is executed at boot. As in the case of traditional malware there exist many different ways to achieve this. For instance, on Windows the executable could simply be placed into the autostart folder. The interested reader can find an overview about common mechanisms that malware leverages to load malicious executables at boot in [144].

On the other side, if the data-only malware used a vulnerability-based infection process, it must ensure the re-exploitation of the vulnerability after each boot. This places additional constraints on the vulnerability that is used for infection. First, the vulnerability must be contained in a program that is executed during boot. This ensures that the malware is loaded with the system. Second, the vulnerability must be *self-triggering*. That is, the vulnerable program must read some external data source, such as a configuration file, which will trigger the vulnerability, load the data-only malware into memory, and start its execution.

However, especially when considering kernel-level data-only malware it is unlikely that the same vulnerability that is used to load the malware is also self-triggering. To solve this problem a *multi-staged* loading process can be used. The main idea behind this approach is to use a second vulnerability to trigger the execution of the infection process. Of course, this second vulnerability must not necessarily be in the same piece of software. As an example, consider a kernel-level data-only malware that uses a vulnerability to load itself that is not self-triggering. To solve this problem we can make use of a self-triggering user space vulnerability to bootstrap the execution of the kernel-level malware. Thereby the user space vulnerability will provide the malware with control over a process that is loaded at system boot. The malware will use this control to actually trigger the kernel-level vulnerability, which loads the kernel component of the malware. Instead of targeting the kernel vulnerability directly, the malware will use a two staged loading process to place itself into the kernel.

Besides leveraging multiple vulnerabilities to conduct a multiple-staged loading process, the attacker can of course also combine file-based infection with vulnerability-based infection to accomplish the same. In the example above, the attacker could, for instance, make use of an executable infected with data-only malware to perform the first stage of the loading process. Once the executable is loaded and starts the execution of the data-only malware, the malware could then once again trigger the kernel vulnerability and load the kernel-space component. This approach has the advantage that the malware does not require a self-triggering vulnerability to function.

Although a multi-staged loading process requires the infection of multiple host applications, it can enable an attacker to overcome various obstacles. By initially leveraging attack vectors that are simpler to exploit, the attacker gains a platform for further exploitation. This might be useful if the second stage of the loading process requires additional information about the running system. This could be the case if the second stage must overcome protection mechanisms such as ASLR, for example. We will provide a concrete example of a resident data-only malware in Section 3.5.4.

3.5 Proof of Concepts & Experiments

So far we have only considered data-only malware from a theoretical point of view. However, to show that data-only malware is a realistic threat and to understand it in all its details it is necessary to analyze the practical aspects of data-only malware as well. For this purpose, we will in this section present detailed proof of concept (POC) implementations of data-only malware. In particular, we will describe a general file-based infection mechanism for ELF binaries and leverage it to create file-based one shot data-only malware in Section 3.5.2, we will use the architecture that we proposed in Section 3.3.4 to construct a persistent kernel-level data-only rootkit in 3.5.3, and we will present a multi-staged loading process for the aforementioned rootkit to demonstrate residence in our third and last POC implementation in Section 3.5.4.

We chose these POCs for two reasons. First and foremost, they demonstrate all of the theoretical concepts that we discussed so far at the hand of practical examples. Most notably, the POC implementations cover all three types of data-only malware as well as both of the infection mechanisms it can use. Second, they prove that data-only malware can be leveraged to construct sophisticated malware capable of infecting recent systems in spite of all modern protection mechanisms. This shows that data-only malware is a realistic threat that can be applied to real world scenarios. To underpin the latter, we begin by describing our test environment.

3.5.1 Test Environment

To provide a realistic attack scenario, we implemented all POCs on a standard installation of a 64-bit Ubuntu 13.04 server VM running Linux kernel version 3.8 with an UEFI BIOS and secure boot enabled. In particular, our attack model assumes the following system/kernel level protection mechanisms:

- UEFI secure boot
- disabled module loading
- disabled `/dev/kmem`

- module ASLR (kernel space)
- stack canaries (user space and kernel space)
- stack reordering (user space and kernel space)
- $W \oplus X$ (user space and kernel space)
- heap protection (user space)
- ASLR (user space)

As one can see, the test environment leverages the typical protection mechanisms that can be found on most systems today and additionally makes use of secure boot. We purposefully chose a 64-bit architecture as this architecture becomes more and more prevalent and makes the implementation of data-only malware on top of that more difficult as arguments must be passed within registers and not on the stack as in the case of a 32-bit architecture. For details about the individual protection mechanisms we refer the reader to Section 2.4.1.

3.5.2 File-based Infection of ELF Binaries

In this section we present a general and stealthy file-based infection approach for the Linux ELF format. Since this file format is in principle very similar to other well-known file formats (e.g. PE), it is likely, however, that the techniques we present can be adapted for the infection of other file formats as well. Note that the technique presented here is, to the best of our knowledge, the first file-based infection mechanism for data-only malware.

As suggested in Section 3.1.3.2, the proposed infection mechanism is based on the Linux loader (`ld`). In particular, we abuse the loader to bypass ASLR by forcing it to construct the control structure at runtime. This is achieved by manipulating the management structures within the binary that the loader leverages to conduct the loading process. By careful manipulation of these management structures, we are able to provide a file-based infection mechanisms that achieves a high-level of stealth and is even hard to detect for experienced users. Since we additionally only make use of gadgets that are contained in the majority of ELF executables, the mechanism is also general applicable. In the following we will describe our file-based infection mechanism in more detail by explaining how we solve the core implementation challenges of data-only malware.

Challenge A: Loading and Storing the Data-only Program. To be able to start the execution of our data-only program later on, we must load it to a predictable memory address in spite of ASLR. We accomplish this by manipulating the *program header table* of the ELF binary, which contains the *segments* of the program as well as their type, load address, size, and access rights [90]. Put simply, a segment is essentially a memory region. During the loading process the loader will parse the program header table and will reserve memory regions based on the segment information contained in it. Consequently, by manipulating a segment entry within the program header table, we can force the loader to reserve a memory area of an arbitrary size at an arbitrary address, which is exactly what we require. In our POC implementation we chose to manipulate the segment entry `GNU_STACK` as this entry exists within every binary compiled with GCC and is on top of that not critical for the execution of most binaries. It essentially defines whether the stack of a program is mapped as executable or non-executable. If the entry is missing, the stack is by default marked as executable.

In the next step, we must find a way to load our data-only program to the memory area that we acquire via the manipulation of the program header table. This can be accomplished by the manipulation of another management structure: the *relocation table*. As the name suggests the relocation table contains all entries that are processed by the loader during relocation. In the case of ELF, a relocation entry is thereby a struct consisting of three values: `offset`, `info`, and `addend` [90]. The first value, `offset`, specifies to what memory address a relocation will be applied. By manipulating this field we can thus write to an arbitrary memory address. This enables us to write data to the acquired memory area. What is still missing is the possibility to resolve the address of the gadgets we want to leverage though. This is where the other values come into play.

The second value, `info`, states the type of the relocation and the index of the symbol to use should the relocation entry refer to one. Every symbol that a binary uses is thereby stored within a symbol table. The index within the `info` value forms the connection between this table and a relocation entry and thus enables us to specify relocation types that leverage symbol information. Most important for us is the relocation type `R_X86_64_64` [89]. When the loader processes such an entry, it will resolve the symbol specified by the symbol index contained within the `info` field and add the address of the symbol to the third and final value, `addend`. The result of this computation will then be written to `offset`. This enables us to resolve the address of arbitrary gadgets using the following approach: First, we require the index of a symbol contained in the same binary as the gadget we want to use. For this purpose, we can either create a new symbol entry or use an existing symbol. Next, we create a relocation entry and set its `info` field to the type `R_X86_64_64` and specify the index of the symbol we selected. Finally, we place the relative offset of the gadget to the symbol we are using into the `addend` field. When the loader process the relocation entry it will then resolve the address of the specified symbol and add the relative offset to it yielding the address of our gadget. This approach works as the relative offset between a symbol and a gadget will always remain the same, since ASLR only randomizes the base address of the code section, however, not the instructions within the code section.

In our POC implementation we leverage the symbol `__libc_start_main` to construct relocation entries. On the one hand, this enables us to resolve the address of arbitrary gadgets residing within the GNU C library (libc). Since the libc has a large codebase that

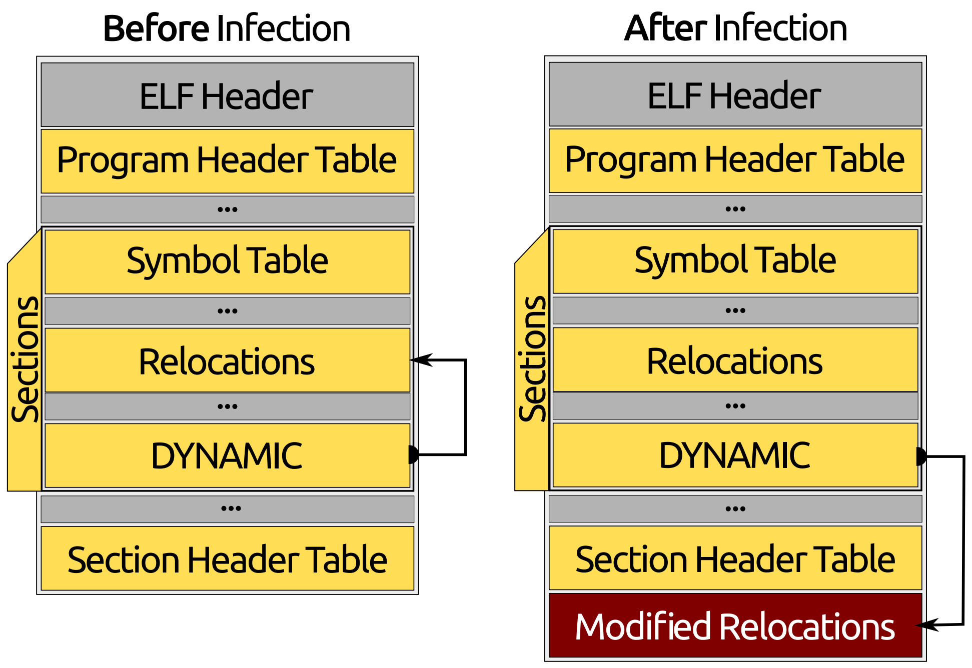


Figure 3.3: Schematic overview of an infected ELF file.

has been proven to be Turning complete multiple times [129, 138], this provides us with a wide variety of gadgets that potentially allow us to perform arbitrary computations. On the other hand, this also increases the generality and stealthiness of the approach, because the libc is an integral part of a Linux system and the symbol `__libc_start_main` is, to the best of our knowledge, contained within every binary that uses the libc. Consequently, we do not have to manipulate the symbol table in any way, while we at the same time can draw on plentiful gadgets shared across Linux systems.

Our final approach for loading our data-only program looks then as follows: First we manipulate the `GNU_STACK` entry within the program header table to create a memory region for our data-only program. In the second step we divide the data-only program into 8-byte blocks. For each block we will create an individual relocation entry. The `offset` field of the first relocation entry will thereby point to the beginning of our memory region. Each entry that follows will increase the address by 8 bytes such that each relocation entry writes directly 8-bytes after the location written by the previous entry. This will write the data-only program block by block into memory. If the block we want to write is supposed to contain the address of a gadget, we will set the type of the `info` field to `R_X86_64_64` and the symbol index to the index of `__libc_start_main`. In addition, we set the value of the `addend` to the relative offset of the gadget to `__libc_start_main`. Otherwise if the block is supposed to contain a constant, we will set the type of the `info` field to `R_X86_64_RELATIVE` and the `addend` to the constant we want to write. This will simply write the value of `addend` into the chain.

Lastly, as one might imagine, this results in quite a few relocation entries which might be noticed by someone inspecting the binary. This is why we leverage another technique to hide the relocation entries from common ELF inspection tools such as `objdump` and `readelf`. Instead of manipulating the original relocation table, we will copy the table as well as our entries to the end of the binary file. Afterwards we will update the `RELA` and `RELASZ` entries within the `DYNAMIC` segment of the binary. As a result, the loader will process our modified relocation table at the end of the binary, while inspection tools will print the *original* relocation table. The problem arises as inspection tools parse the relocation section within the binary. The loader on the other side leverages the address specified in the `DYNAMIC` segment. The final layout of an infected binary is visualized in Figure 3.3.

Challenge B: Starting the Data-only Program. Once the the data-only program has been loaded, we must activate it. That is, we must place the SP to the location of our data-only program and execute a `ret` instruction. For this purpose we create two more relocation entries. However, in contrast to the previous entries, these will not be used to write into the memory region of the data-only program. Instead, they will be leveraged to overwrite the value of a variable and the address of a function. To understand this approach we first have to take a closer look at the execution of an ELF program.

Once the loader finishes the loading process, it will invoke the entry point of the binary. In contrast to popular belief, this is usually not the `main` function of a program, but a special function named `_start`. The `_start` function is essentially responsible for conducting any initialization tasks that must be performed before the `main` function can be invoked. Most notably, the `_start` function will invoke the function `__libc_start_main`, which will perform the initialization of the libc. During this initialization procedure `__libc_start_main` will call back into the binary and invoke the function `__libc_csu_init` to start the execution of any constructors the binary may have [65]. In the process, `__libc_csu_init` invokes `_init` which contains the following code:

Listing 3.3: The `_init` function of ELF binaries.

```
1  sub   rsp,0x8
2  mov   rax,QWORD PTR [rip+0x200b3d]     # 600ff8 <_DYNAMIC+0x1d0>
3  test  rax,rax
4  je    4004c5 <_init+0x15>
5  call  400520 <__gmon_start__@plt>
6  add   rsp,0x8
7  ret
```

As we can see, the function loads a variable from the location `_DYNAMIC+0x1d0` into `rax` (Line 2) and afterwards invokes `__gmon_start__` (Line 5) given that `rax` is not zero. We can abuse this code fragment to perform a stack switch. For this purpose, we will first create a relocation entry that overwrites `_DYNAMIC+0x1d0` with the address of our data-only program. In the second step, we create another relocation entry that overwrites the address of `__gmon_start__` within the GOT with the address of a `xchg eax,esp; ret;` gadget. The first overwrite will thereby ensure that the address of our data-only

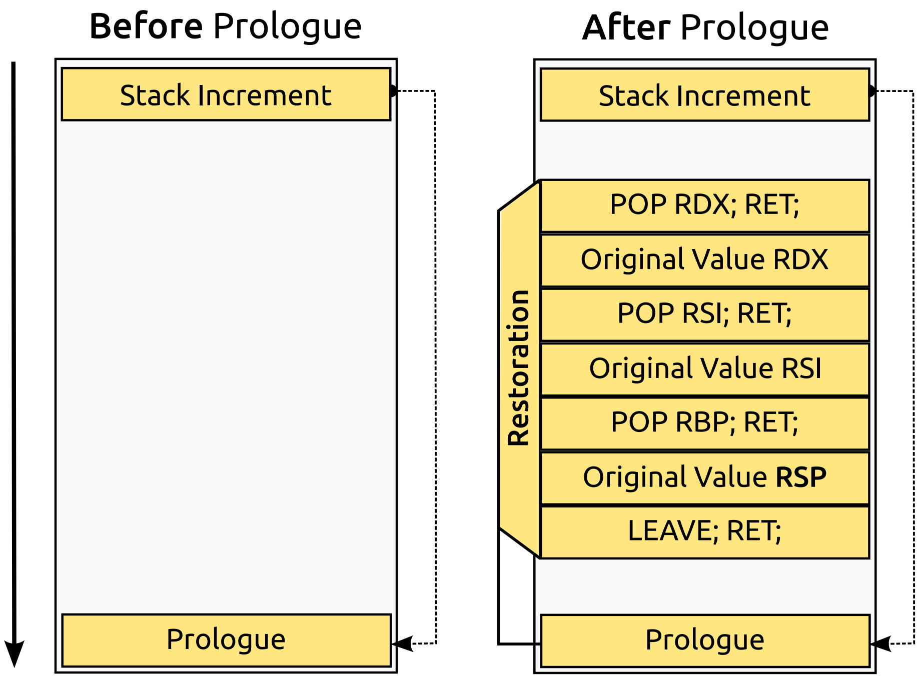


Figure 3.4: Overview of the restoration control structure that is created during the execution of the prologue control structure.

program is loaded into `rax` (Line 2) . Since `rax` will be greater than zero in this case, the `_init` function will try to invoke `__gmon_start__`. Since we overwrote its address in the GOT, the call will, however, not invoke `__gmon_start__`, but our switching gadget `xchg eax,esp; ret;`. The only requirement to leverage this technique is that our data-only program resides at a 32-bit address, since `xchg eax,esp; ret;` will clear the upper 32-bits of `rax` during the exchange. This is not a problem, however, as we can freely choose the memory address of our data-only program as we previously described.

In summary, we can start the execution of our data-only program by creating two relocation entries that will overwrite the value of `_DYNAMIC+0x1d0` with the address of our data-only program and the address of `__gmon_start__` within the GOT with the address of a `xchg eax,esp; ret;` gadget. Since `xchg eax,esp; ret;` instructions occur frequently within code, this leads to widely applicable activation mechanism. In addition, because the overwrites are performed by the loader at runtime and the relocations are not visible within tools such as `objdump`, the proposed approach is also quite stealthy.

Challenge C: Restoring a Valid Execution Path. After our data-only program has been executed, we must resume the normal execution of the infected binary. To accomplish this we make use of a *prologue* and an *epilogue* control structure. The prologue control structure is executed directly after we perform the stack switch. It will back up important registers and create a *restoration* control structure that can then be invoked by the epilogue to restore the original values of the backed up registers and to resume the original control flow. To create the restoration control structure, the prologue control structure will first move the current SP into `rax`, which does not contain any crucial information and can thus be overwritten, and then create some room on the stack by adding a constant offset to the SP. The resulting space, which is pointed to by `rax`, is then used by the prologue control structure to create the restoration control structure. This structure is shown in Figure 3.4.

As one can see, the restoration structure will first restore the value of `rdx` and `rsi`, which contain the environment pointer and the argument pointer, respectively. Next, the restoration control structure will load the value of the *original* SP, which can be deduced from `rdx` or `rsi`, into the `rbp` register. Finally, it will leverage a `leave; ret;` gadget, which essentially moves the FP into the SP, to switch back to the original stack. The `ret` following the `leave` will thereby load the return address of the `_init` function into the IP thus resuming the original control flow.

Once the prologue has been executed and the restoration control structure has been written, the data-only malware is free to use any register to perform its computations with the exception of `rbx`. The reason for this is that the prologue will store the address of the restoration control structure in this register. The epilogue will then load the address contained within `rbx` into `rbp` and execute a `leave; ret;` gadget, which will lead to the execution of the restoration chain and the execution of the original execution path.

Experiments To verify the generality of the proposed approach, we leveraged our file-based infection mechanism to infect 20 Linux programs contained in the `/bin/` and `/bin/ls` directories including `ls`, `more`, `rm`, `cat`, `ps`, `gcc`, and `vim` with a simple one shot malware that prints a message to the screen and then exits thereby resuming the execution of the original program. As expected, the approach worked flawlessly with all of them. In addition, to prove that we are also able to infect very small programs, we created a simple "Hello World!" application and applied our infection approach. As in the case of the other binaries, the program could successfully be infected even though it actually only provides a very small codebase. Consequently, the proposed mechanism is indeed quite general.

Summary. In this section we presented a file-based infection mechanisms for Linux ELF binaries. The main idea behind the mechanism is to force the loader to create our data-only malware during runtime by carefully manipulating the management structures within the binary that the loader uses. In particular, we manipulate the program header table to reserve memory, the relocation table to load and activate our data-only program, and the DYNAMIC array to hide the modified relocation table. In the process, we solely leverage gadgets from the libc or parts of the binary that are related to the libc and are thus part of most Linux ELF binaries. Due to this approach, the resulting infection mechanism is quite general and is able to reliably infect all of the ELF binaries in our experiments with one shot data-only malware. Since we additionally leverage various mechanisms to hide our tracks, the infection mechanism is difficult to detect in practice.

3.5.3 Persistent Kernel-Level Rootkit

To demonstrate the feasibility of the concepts discussed in Section 3.3 we implemented a persistent data-only ROP rootkit. We chose to implement our POC in the kernel as this demonstrates an especially dangerous form of malware. However, we argue that the concepts outlined can easily be leveraged to infect a userland process as well.

For our POC we assume a local attacker that has user-level access. Further we presume a vulnerability in kernel space which enables us to load our rootkit. To provide a realistic attack scenario, we used a real Linux kernel vulnerability for this purpose. It is interesting to note that generally one assumes that an attacker has root privileges at the time she is ready to install a rootkit. However, since loading our rootkit requires a vulnerability, it can be loaded without requiring root privileges.

Before we go into the details of our implementation, we begin with a quick overview. Our POC is designed according to the architecture presented in Section 3.3.4. The initialization stage and the persistent stage are further divided into four ROP-chains. The initialization stage is only composed of a single ROP-chain, the initialization chain. The initialization chain is only executed once during initial exploitation and its single purpose is to setup the execution of the persistent stage. The persistent stage on the other side is composed of three different ROP chains, the copy chain, the dispatcher chain, and the payload chain. The copy chain is thereby the only truly persistent chain. It is invoked whenever the hooks the malware placed into the system are triggered. On every invocation the copy chain builds a dispatcher chain in a predefined memory area. The dispatcher chain will then in turn create a unique state and a unique payload chain for each process. The payload chain provides the actual functionality of the rootkit. In the case of our POC, the rootkit hooks the read and getdents system call to provide key logging, process hiding, and file hiding. We chose to implement these mechanisms, to demonstrates that persistent data-only rootkits can indeed provide functionality similar to traditional rootkits.

Initialization Stage. The initialization phase in a kernel rootkit requires a vulnerability in the kernel code. We used the real Linux kernel vulnerability CVE-2013-2094[3] for this purpose. This vulnerability essentially allows a user space application to take control of a pointer variable within the kernel. With the help of this pointer variable the application can increase the value of memory words within kernel space. By increasing the address of an interrupt handler, the handler can be made to point to a `leave; ret;` gadget in

[3]http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2094

the kernel. The `leave` instruction moves the current FP into the SP and then pops the current value on top of the stack into the FP (`mov rsp, rbp; pop rbp;`). By placing the address of the initialization chain into the FP, we can use this gadget to start the execution of our initial ROP-chain. For this scheme to work, we have to trigger the interrupt handler that we modified by provoking an exception or executing an `int X;` instruction, where X is the number of the interrupt whose handler was changed. In addition, we have to setup the FP to point to our initial ROP-chain, which will then be loaded into the SP by the `leave` instruction. Since the attacker triggers the exploit, this is not a problem in this scenario. Notice, however, that this stack pivoting mechanism is only possible as we control other registers besides the IP (the FP in this case). The triggering of a hook on the other side is *not* controlled by an attacker, which implies that we cannot simply use such a technique as a switching mechanism. This demonstrates the difference of stack switching in the case of exploits and hook invocations at a practical example.

Once the initialization chain gains control, it will allocate three memory areas within kernel space. First, memory for the global state is allocated. The address of the state is then patched at runtime into every location where it is used within the copy chain and the dispatcher chain. In the next step, memory is allocated for the dispatcher chain. The address of this memory area is then once more patched into a predefined location within the copy chain such that the copy chain can directly use it during its execution. This step is necessary, since the copy chain needs to copy the dispatcher chain into this memory region on every invocation. Notice that the copy chain cannot simply allocate memory as this would involve an external function call that would overwrite parts of the persistent chain. Finally, memory for the copy chain is allocated and the copy chain is copied into this memory area.

At this point the initialization chain sets up our stack switching mechanism. In this case, we use the `sysenter` mechanism described in Section 3.3.2.1. This means that our initialization chain must write the correct values in the `sysenter` MSRs. Specifically, the address of a `ret` instruction is written into the `IA32_SYSENTER_EIP` MSR and the address of the copy chain is written to the `IA32_SYSENTER_ESP` MSR. Finally, the hooks are set by overwriting the read and the getdents system call in the system call table with the address of a `sysenter` instruction.

Persistent Stage. Once the initialization phase is completed, the hooks are set and the persistent stage is waiting to be triggered. The triggering of the hooks is illustrated in Figure 3.5. As the read and the getdents system call function pointer was overwritten in the initialization stage, any call to the read or getdents system call (1) will result in the `sysenter` instruction being called (2). As described in Section 3.3.2.1, the `sysenter` instruction will load the new IP from the `IA32_SYSENTER_EIP` MSR and the new SP from the `IA32_SYSENTER_ESP` MSR (3). Since we overwrote these MSRs, our copy chain is executed (4), thus addressing the challenge outlined in Section 3.3.1.4. It is important

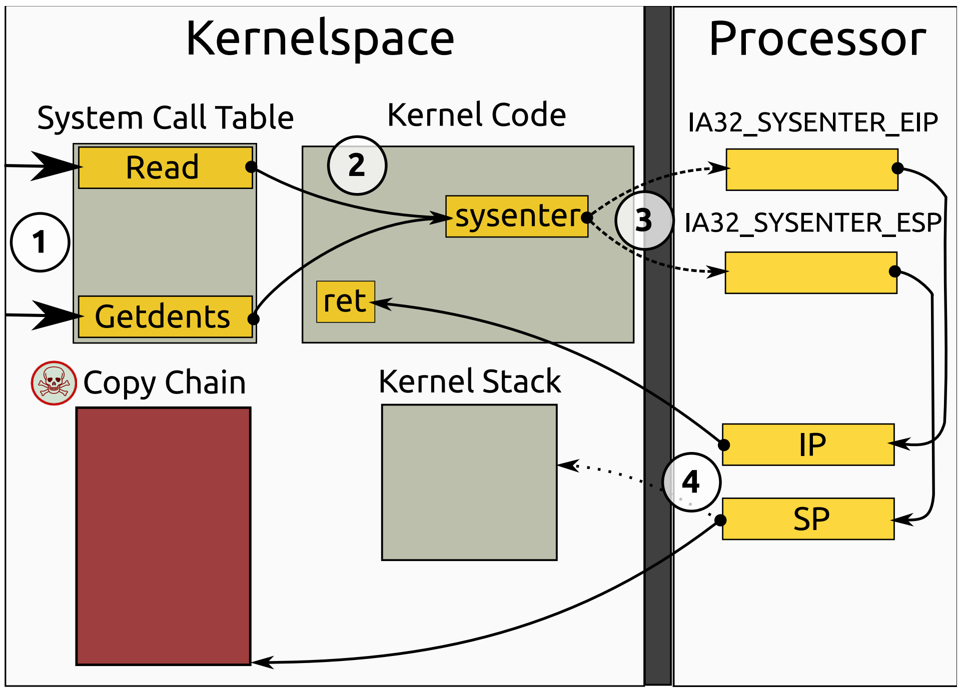


Figure 3.5: Overview of the sysenter hook on the read and getdents system call that is used by our POC to permanently alter the control flow of the system.

to note that the `sysenter` instruction is our mechanism for switching the stack and is completely *independent* of the fact that we are hooking system calls.

When a hook is triggered, it invokes the switching mechanism, which in turn invokes the persistent stage of the rootkit. As outlined in Section 3.3.4, interrupts must be disabled when the persistent stage of the malware is invoked to avoid interrupt-induced overwrites. In our POC, this is achieved with the help of the `sysenter` instruction that automatically disables interrupts when it is executed. Notice that both of the proposed hardware-based switching mechanisms provide this feature. Should a software-based switching mechanism be used where interrupt-induced overwrites are an issue (e.g. in kernel space), the gadget that performs the stack switch must also disable interrupts.

An overview of the persistent stage of the POC is shown in Figure 3.6. The first chain that is activated in the persistent stage is the copy chain. The first important task that this chain performs is to save the current state of the CPU (1) such that control can be gracefully restored after the payload has executed. The copy chain stores the values of critical registers in the global state (2). To use this approach, the initialization chain patches the addresses of the global state into the copy chain before its first execution.

After the registers have been saved, the copy chain copies the dispatcher chain (3) into the memory area that has been preallocated for it by the initialization chain. As soon as this copy operation is completed, execution is transferred to the newly created dispatcher chain (4). The dispatcher chain can now use all of the registers saved by the copy chain.

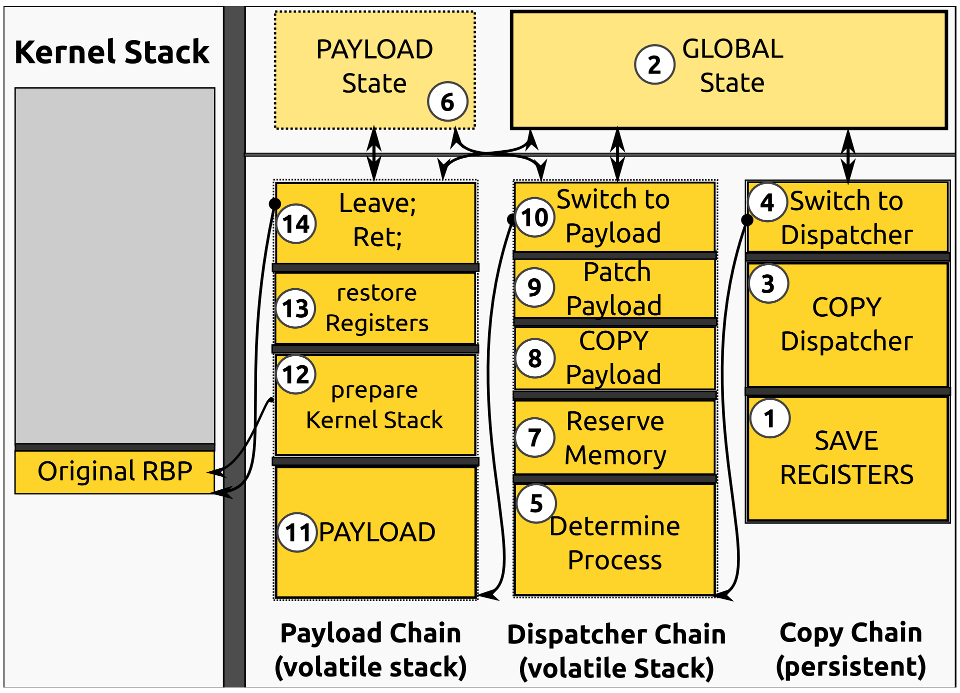


Figure 3.6: Overview of the persistent stage of the rootkit and individual chains used in this stage.

The dispatcher chain starts by obtaining the current process data structure (5). To do this we make use of a Linux data structure that is associated with each CPU, the `per_cpu` data structure. This data structure contains, among other things, a pointer to the `task_struct` structure of the process that is currently executing. The rootkit obtains this pointer and uses it to locate the state (6) for this specific process. For this purpose, the global state contains an array that stores a pointer to the state of each process together with the address of it's `task_struct` pointer. By searching this array for the `task_struct` pointer of the process, the rootkit can thus obtain the address of the process's state area. If a state does not yet exist for the current process, the dispatcher chain will allocate a new memory region for the state and store it together with the `task_struct` pointer in the array.

Once the address of the state of the process is known, the dispatcher chain will copy the values of the registers that have been saved by the copy chain (and currently reside within the global state) into the process state. In the next step, the dispatcher will allocate a new memory area for the payload chain of the process (7) and copy the payload chain into the newly allocated region (8). Finally, the dispatcher must patch (9) all the locations in the newly created payload chain that refer to the process state (6) or the global state (2) at runtime.

After the payload has been patched, it is ready for execution and the dispatcher will transfer the control to the newly created chain (10). At this point the payload chain is free to enable interrupts as the payload is now unique for each process. The payload (11) will then execute the desired functionality. In the case of our POC this functionality consists of key logging, process hiding, and file hiding. To achieve the former the rootkit will copy every character that is typed by the user into a buffer within the process state. Data typed by the user is thereby identified based on the file descriptor that is specified in a read system call. As soon as the user types a return character, which marks the end of a command, the data in the process state is interpreted. When the data within the state corresponds to a specific rootkit command such as hiding a specific process, the rootkit will execute the command and delete the data within the buffer. Otherwise the rootkit will write the data entered by the user to the kernel log. Thus the attacker is able to control the rootkits behavior from the command line.

Process hiding is realized by setting the PID of the process that should be hidden to zero. As a result, the process will no longer be displayed by programs such as `ps`. The process that is to be hidden (or unhidden) can thereby be specified using the above described communication mechanism. File hiding on the other hand is realized by intercepting the getdents system call and removing any entries from the returned structures that should be hidden. The filename of hidden entries must thereby begin with a predefined string. Since these are well-known techniques, we will not describe them in more detail at this point. Notice that the payload can distinguish between read and getdents system calls based on the system call number.

At the end of the payload chain, the original execution path must be restored and control must be handed back to the kernel. In our POC implementation we make use once more of a `leave; ret;` gadget (14) for this purpose. By placing the value of the kernel SP into the FP, we can switch the SP back to the original kernel stack using this instruction. However, deducing the original value of the kernel SP remains an issue as the SP was overwritten the moment our copy chain assumed control through the execution of the `sysenter` instruction. To solve this problem, we once more rely on the `per_cpu` data structure, which also contains the value of the SP at the time when the kernel assumed control from user space. By reading this value and subtracting the stack frame size of the system call handler (the function that is executed immediately before our hook is invoked), we can calculate the value the SP had before the `sysenter` instruction was invoked. We were forced to use this approach as the FP was not set in our case (i.e., we find ourselves at the bottom of the stack). Usually, however, it is possible to use the FP in the same manner we use the saved SP in the `per_cpu` data structure. That is, one can subtract the size of the current frame from the FP to obtain the original value of the SP. Notice that this method is very general and also useful in userland exploitation.

To make use of a `leave; ret;` gadget, we first have to prepare the kernel stack for the switch (12). Since the `leave` instruction will move the FP to the SP and then pop the current value on top of the stack into the FP, we have to copy the original FP onto the kernel stack. This will ensure that the original FP is loaded, when we hand control back to the kernel.

Finally, there is one last step we must take to ensure a smooth transition back to the original execution flow. We have to restore the original register values (13) that were saved by the copy chain. These registers values are currently stored within the payload state. Before the `leave; ret;` instruction is executed, the stored values are transferred back into the corresponding registers. The only exception is the FP. Since the FP is used by the `leave` instruction, it will be loaded with the address of the kernel stack pointer. The original value of the FP will than be restored in the process of the stack switch as has been described above.

Experiments. We leveraged our rootkit to infect our test system. By exploiting the kernel vulnerability, the rootkit managed to load itself in spite of all the protection mechanisms detailed in Section 3.5.1. Once loaded, we tested the rootkits functionality by enabling and disabling key logging as well as hiding and unhiding multiple files and processes. Throughout our experiments the system remained stable and we did not experience any side effects. In spite of logging key strokes, the system did not show any noticeable increase in response time.

Summary. In this section we showed that the proposed architecture for persistent data-only malware can indeed be leveraged to implement sophisticated data-only malware such as a kernel rootkit in practice. Instead of utilizing a file-based infection approach, we in this case chose to use a vulnerability-based infection approach to demonstrate that this attack vector is still available to data-only in spite of modern protection mechanisms. To make the attack scenario as realistic as possible, we selected a real kernel vulnerability to load the kernel rootkit into memory. The rootkit itself is thereby capable of key logging, process hiding, and file hiding and thus provides similar functionality as code-based rootkits. To achieve this, the rootkit must place multiple hooks within the system call table. This shows that the proposed architecture for persistent data-only malware supports the usage of multiple hooks as well as multiple threads and is thus well suited for the use in real world scenarios.

3.5.4 Residence

The last concept that we have not covered from a practical perspective so far is residence. To show that residence is possible, we created a multi-staged loading process for our kernel rootkit that we presented in the last section. We thereby implemented the loading process using a vulnerability-based infection mechanism. For this purpose, we first implemented a simple C program that prints a “quote of the day”. To obtain the database of quotes, which is a plain text file, it reads a configuration file that tells it where the database is stored and how many quotes are contained in it. Due to an error in the function that parses the configuration file, the attacker can, however, overwrite the saved IP on the stack and divert the control flow.

To realize the multi-staged infection approach, we placed the binary representation of the rootkit into a text file and manipulated the configuration file to trigger the vulnerability together with a small loader chain. This loader chain essentially fulfills two tasks: first, it loads the rootkit into memory by using the same function that the vulnerable program uses to load the configuration file and the file containing the quotes. Second, it triggers the vulnerability and uses it to activate the initialization chain of the rootkit. Once the initialization chain takes control, the normal loading process occurs as has been described in the last section. For the sake of simplicity, we added the gadgets required to trigger the kernel vulnerability into the binary. We like to stress, however, that these gadgets could similarly be obtained from the libc.

To test our approach, we configured the test system to execute the vulnerable program at boot. Once the program was executed, it triggered its self-triggering vulnerability by loading the manipulated configuration file. This led to the execution of the loader control structure, which in turn loaded and executed the rootkit.

While we simplified the implementation process in this case, the example shows that residence is possible given that the host application (or one of its libraries) contains the necessary gadgets. Due to the fact that we implemented an entire rootkit solely using the instructions within the Linux kernel binary, we argue that this is likely the case in practice especially as our file-based infection approach, which we could similarly use to implement residence, provides access to the entire libc. We will discuss the problem of computability in the case of data-only malware in more detail in the Section 3.6.1.

3.5.5 Summary

We provided three POC implementations that validated the theoretical concepts we have discussed so far using practical examples. In particular, our first POC presented a general and stealthy file-based infection mechanism for ELF binaries which we leveraged to infect various binaries with one shot data-only malware. In our second POC we made use of a real kernel vulnerability to demonstrate a vulnerability-based infection mechanism under realistic circumstances and showed that data-only malware can indeed be used to implement sophisticated persistent malware, such as kernel-level rootkits, that is in no way inferior to sophisticated traditional malware. Finally, our third POC demonstrated that data-only malware can leverage a multi-staged loading process combined with a self-triggering vulnerability to achieve residence. In summary, these examples show that data-only malware is a realistic and dangerous threat in practice.

3.6 Discussion

By discussing the infection mechanisms and execution strategies that data-only malware can employ in theory and in practice, we established a solid understanding of the malware form and the challenges associated with its creation. However, besides the elementary properties of infection and execution, we also introduced advanced properties of malware in Section 2.1.4, that, while not required for its functioning, are essential for malware to achieve its objectives. In this section, we will thus continue our analysis of data-only malware by considering these properties. To this end, we will first discuss the computational ability of data-only malware, before we consider its level of stealth, its environment dependencies, and its capability to evade signature-based detection.

3.6.1 Computability

One of the key questions that must be answered when considering data-only malware is whether this malware type has the same computational ability as traditional malware, given that it relies on code reuse to function. To provide a general, practice-oriented answer to this question, we will consider two individual aspects of code reuse: first, what is the likelihood that an application *provides* a Turing complete gadget set? Second, if an application provides a Turing complete gadget set, do we have *access* to this set when implementing data-only malware in spite of the modern protection mechanisms described in Section 2.4.1? In the following, we will address both of these questions individually starting with the former.

Turing Completeness in Practice. As discussed in Section 2.2.2, a single proof of Turing completeness is only of very limited value when we want to determine the computational ability of code reuse techniques in general. Instead, we must find a way to estimate the likelihood of encountering an application that provides a Turing complete gadget set in practice. To accomplish this, we require a large scale real world analysis of applications and the gadget sets they provide. Homescu et al. [60] performed such an analysis for Linux based OSs.

To be able to consider a wide range of applications in their analysis, Homescu et al. used a two-staged approach. In the first step, they created a hand-picked gadget set solely consisting of gadgets with a maximum length of 3 bytes. The simple reason for this selection being that a small number of bytes is more likely to appear within a binary compared to a large number of bytes. In the second step, they then divided the hand-picked set of gadgets into *classes*. The selection of the classes was thereby conducted in such a way that the existence of a single representative of each class would be sufficient to form a Turing complete language. Instead of testing for a single set of Turing complete gadgets at a time, this enabled them to test all possible Turing complete gadget sets contained in their hand-picked list in a single pass.

In their analysis Homescu et al. tested all applications contained within the `/usr/bin` folder on eight different OSs. In the process, they used two different scanning modes. While they only scanned the binary itself for Turing completeness in the first mode, they additionally scanned the libraries that a binary uses in the second mode. The results of the experiments are show in Table 3.1.

On average, they tested 1860 applications per OS. Even if we ignore OpenSUSE, which is an outlier in the statistic, nearly 35% of these applications provided a Turing

Distribution	**Binaries**	**TC Libraries (A)**	**TC Binary (B)**	**∅ (A)**	**∅ (B)**
CentOS 6.0	2231	783	20	35.10%	0.90%
OpenSUSE 11.4	2292	1804	77	78.70%	3.36%
PCLinuxOS ’11	2405	955	56	39.71%	2.33%
Fedora 15	1881	758	39	40.30%	2.07%
Kubuntu 7.10	1337	404	27	30.22%	2.02%
Kubuntu 11.10	1655	565	45	34.14%	2.72%
Ubuntu 9.04	1492	434	31	29.09%	2.08%
Ubuntu 10.04	1587	497	35	31.32%	2.21%
Total	**14880**	**6200**	**330**	**41.67%**	**2.22%**
w/o OpenSUSE	**12588**	**4396**	**253**	**34.92%**	**2.01%**

Table 3.1: Overview of Linux binaries providing a Turing complete (TC) gadget set Linux on various distributions. The first column shows the number of tested binaries. The second and third column show the number of Turing complete binaries with (A) and without (B) libraries. Adapted from [60].

complete gadget in the second scan mode (binary and libraries). However, when scanning the binary on its own, only 2% of the binaries provided a Turing complete gadget set. This shows that access to libraries is critical for achieving Turing completeness, which intuitively makes sense, since binaries are often small compared to the libraries they utilize. In addition, libraries are often shared among applications, which means that a single Turing complete library can affect many different applications. In fact, this could be one of the reasons why the number of Turing complete applications in the case of OpenSUSE is almost twice as high as for the other OSs when libraries are considered (79%), while it is only slightly higher without libraries (3%).

Another important observation that can be made based on the results shown in Table 3.1 is that the newer the OS version, the higher the amount of executables providing a Turing complete gadget set. For instance, in the case of Kubuntu 7.10 the amount of binaries containing a Turing complete gadget set was 30%, while it was 34% on Kubuntu 11.10. This also makes sense as the codebase of binaries is likely to grow with time [182]. Consequently, newer systems will probably provide a larger attack surface than older systems.

In summary, the extensive experiments conducted by Homescu et al. show that a considerable amount of applications provides the attacker with a Turing complete gadget set given that she can access the code section of the binary as well as its external libraries. In addition, since codebases tend to get larger with time, it is likely that the situation will even improve for the attacker in the future.

Circumventing Protection Mechanisms. In the last paragraph we established that about 35% of all applications provide a Turing complete gadget set for code reuse.

However, this is only the case if an attacker is able to *access* the binary as well as its libraries. Consequently, a second aspect that we have to consider with regard to the computational ability of data-only malware is whether the attacker has access to this codebase given modern protection mechanisms such as ASLR.

To address this question, we must consider the two different infection mechanisms that data-only malware can employ. The first infection mechanism that we considered was vulnerability-based infection. To achieve Turing completeness in this case, we must find a control flow modifying vulnerability that provides us with access to the binary as well as its libraries. While exploitation has become more difficult over the last years, it is a reasonable assumption that such vulnerabilities exist. For example, according to CVE details [35], 19.8% of all reported vulnerabilities in the year 2014 were code execution vulnerabilities. That is, every fifth vulnerability discovered last year could potentially be used to load data-only malware. It is therefore quite likely that an attacker can leverage this attack vector in practice.

The second infection mechanisms that we considered within this thesis is file-based infection. As we presented a general file-based infection mechanism for ELF binaries that provides access to the executable as well as its libraries, it is very likely that an attacker can obtain access to a Turing complete gadget set using this attack vector.

Summary. In conclusion, in spite of relying on code reuse, data-only malware can in many practical cases perform Turing complete computations. This makes data-only malware to a powerful and realistic threat that can provide a basis for sophisticated attacks. Since the codebases of systems will continue to grow, data-only malware will even become more powerful in the future.

3.6.2 Stealth

An important property of malware is *stealth*. In particular, we are interested in the fact of whether data-only malware can in general be considered more stealthy than traditional malware, because the more stealthy a malware type is the harder becomes its detection. To answer this question, we will classify data-only malware based on the stealth malware taxonomy of Rutkowska [131] that we described in Section 2.1.4.4.

Data-only malware only changes *dynamic* system resources (data). According to the stealth malware taxonomy, data-only malware must therefore be considered as “type 2 malware”. However, this classification would not distinguish data-only malware from traditional type 2 malware, even though there is a fundamental difference between both malware forms. While data-only malware *exclusively* changes dynamic system resources, traditional type 2 malware changes the existing codebase of the system by introducing new instructions. Since code is considered as a *constant* system resource according to Rutkowska model, this means that traditional malware can actually never be type 2 malware. By definition, traditional malware must always change the codebase. Consequently, real type 2 malware did not even exist when Rutkowska presented her stealth taxonomy in 2006. Instead, the malware that Rutkowska considered to be type 2 malware must actually be seen as type 1 malware. In other words, data-only malware is the only form of type 2 malware.

This observation implies that data-only malware is by definition more stealthy than traditional malware and consequently harder to detect. The only exception to this rule represents type 3 malware. However, the stealth of this malware form is based on the properties that virtualization provides and not on the implementation approach of the malware. That is, the stealth of this malware variant will depend on the fact of how well the malware can hide the virtualization layer, whether the malware is implemented using code or data is thereby irrelevant as the guest is unable to access the hypervisor directly.

3.6.3 Environment Dependencies

In Section 2.1.4.5 we established that malware in general is unable to infect arbitrary systems, but instead requires a specific environment to function. These *environment dependencies* are interesting as they could potentially provide a basis for countermeasures, especially if the dependency is inherent to the malware type and cannot be easily resolved. In this section, we will thus discuss the environment dependencies of data-only malware in more detail. This discussion will provide the basis for the countermeasures that we will present in Chapter 7.

We considered seven different important environment dependencies that heavily affect the capabilities of traditional malware. Namely, these dependencies include *hardware architecture dependency*, *operating system dependency*, *operating system version dependency*, *application dependency*, *file format dependency*, *vulnerability dependency*, and *network dependency*. In the following we will analyze how each of these dependencies affects data-only malware. For this purpose we first discuss the dependencies that are shared between traditional malware and data-only malware, before we cover the differing dependencies as well as the additional dependencies that data-only malware has.

Shared Dependencies. Naturally, some of the aforementioned dependencies are inherent to both traditional malware and data-only malware. Traditional malware is in generally bound to a specific hardware architecture that is capable of executing the instructions it is made of. Data-only malware essentially consists of a control structure containing pointers. These pointers point to instructions that eventually must be executed by the underlying hardware. Consequently, data-only malware has the exact same *hardware architecture dependencies* as traditional malware.

Additionally, similar to traditional malware, data-only malware in general depends on a particular OS. The reason for this is that this dependency is not inherent to the concept of data-only malware (or traditional malware for that matter), but is rather dependent on the functionality that malware provides. As soon as this functionality requires OS features (e.g. system calls), the malware will have an *operating system dependency* irrespective of its form. The same applies to *operating system version dependency* and *network dependency*, which are also not bound to the malware form, but to the malware's functionality.

Differing Dependencies. The most significant difference in dependencies between traditional malware and data-only malware is *application dependency.* While some traditional malware instances may be application dependent (e.g. malware written in Java), data-only malware *always* depends on an application to function. This is due to the fact that data-only malware is based on code reuse. Consequently, it is always dependent on the application (or the set of applications) whose instructions it uses to perform its own computations. In case of data-only malware application dependency is thus comparable to hardware architecture dependency.

Besides requiring an application for *execution*, data-only malware also depends on a host application for *infection.* This host can thereby either be the same application it uses for its execution (code reuse) or a different application. Further, since the infection can either be conducted using a file or a vulnerability, data-only malware is *always* dependent on a specific file format or a vulnerability. The latter is in general more common.

Traditional malware only shares this *file format dependency* or *vulnerability dependency* marginally. In the simplest form, the victim is tricked into executing the malware directly. This implies that traditional malware does not require an existing application as host for infection. Additionally, the malware only partially depends on a file format. In fact, it can use any file format that can be executed by the victim's OS. Data-only malware on the other side must manipulate the management structures of a file format for file-based infection, which couples the malware tightly to a particular format.

Additional Dependencies. Besides the dependencies we covered so far, data-only malware faces additional dependencies that traditional malware does not have. These dependencies are a direct result of the implementation challenges we discussed in Section 3.1.2. First and foremost, data-only malware depends on a control structure. The control structure contains the pointers that define the computations that the malware performs. These pointers can be seen as the "instructions" of the malware. The resulting *control structure dependency* is thus comparable to the dependency on the code of traditional malware.

Second, data-only malware depends on a *switching sequence* that activates the execution of the control structure. The switching sequence can be compared to the entry point of traditional malware. It effectively sets the virtual IP to execute the "instructions" within the control structure. The switching sequence itself thereby depends on the code reuse technique that the malware is based on. In case of ROP, for instance, the switching sequence must set the SP to point to the control structure. In the simplest case this can be achieved by directly writing the control structure onto the current stack. The switching sequence then simply consists of a `ret` instruction.

3.6.4 Encryption, Polymorphism, and Metamorphism

Signature-based detection is one of the most widespread approaches to malware detection. As such, the question arises whether data-only malware can similarly to traditional malware leverage encryption, polymorphism, and metamorphism to evade this detection approach or if signatures are an effective remedy against this malware type. As it turns out, all of the techniques are similarly applicable to data-only malware. In this case, however, the techniques are applied to the control structure of the malware and thus to data instead of code. In the following we will take a closer look at encryption, polymorphism, and metamorphism and their advantages and disadvantages in the context of data-only malware. Before doing so, however, we will briefly cover the two types of signatures that can be created for data-only malware.

When considering signatures for data-only malware, these signature could either be created for the "instructions" (i.e. pointers) that the malware consists of or the code that is executed on behalf of the malware. To achieve the latter, we could, for instance, monitor all instructions that are executed on a system and try to identify sequences belonging to data-only malware. While this approach has a large overhead, it can be very effective. From here on we will refer to this kind of signatures as *code signatures*. On the other side, we can try to create a signature for the control structure of the malware and the "instructions" contained within the control structures. For the remainder of this thesis we will refer to such signatures as *control structure signatures*. With this knowledge in mind, let us take a look at encryption, polymorphism, and metamorphism and how these techniques work in the case of data-only malware.

Encryption. Encrypted data-only malware operates similar to encrypted traditional malware: Before the malware is placed into memory, its execution control structure is processed with an encryption algorithm using a secret key. After the encryption a small decryption control structure is perpended to the encrypted malware. Once the execution of the malware begins, the decryption control structure will then decrypt the execution control structure in memory and start the execution of the actual malware. Similar to all operations that data-only malware performs, a requirement for this approach is that the target system provides the instructions necessary for decryption.

While encrypting the control structure of data-only malware can be effective against control structure signatures, it cannot protect the malware against code signatures. This is due to the fact that the instructions that will be executed by the hardware always remain the same independent of the encrypted control structure. In addition, the encryption will only thwart control structure signatures as long as it remains encrypted. Similar to code, the control structure must, however, be decrypted at some point in time during execution. At this point the decrypted control structure can be detected in memory. Finally, it might still be able to create a control structure signature for the decryption routine. Consequently, encrypted data-only malware has the same drawbacks as encrypted traditional malware.

Polymorphism and Metamorphism. Polymorphism and metamorphism require a mutation engine that changes the form of the control structure. In Section 2.1.4.6 we discussed five different techniques that a mutation engine can use to achieve this: *junk code insertion*, *instruction replacement*, *instruction permutation*, *variable/register substitution*, and *code transposition*. In the following we will consider each of these mechanisms in turn.

Junk Code Insertion Junk code insertion in the context of data-only malware is effectively accomplished by adding random data to the control structure. To compensate for this data, additional instructions must be inserted that remove the junk data during execution. For instance, in the case of ROP we could add arbitrary data to the control structure and account for this change by increasing the SP right before the junk data begins by the size of the junk data. This approach can be effective against control structure signatures as well as code signatures, since it will change both the control structure and the executed instructions.

Instruction Replacement There are two possible approaches to instruction replacement for data-only malware. On the one hand, we can use the same approach that traditional malware uses and replace gadgets with different but semantically equivalent gadgets. This will affect control structure signatures as well as code signatures.

On the other hand, if there exist multiple copies of the same gadget in memory, we can replace a pointer within the control structure with a different pointer pointing to the exact same gadget. While this will not allow us to evade code signatures, the approach is very simple and may allow us to evade control structure signatures.

Instruction Permutation The order of the pointers in the control structure of data-only malware can be changed as long as the resulting computation remains the same. By changing the order of the pointers we will change the layout of the control structure as well as of the order of the code that will be executed.

Variable/Register Substitution Changing the memory addresses of variables can in the case of data-only malware often be achieved without altering the gadgets that the malware uses. The reason for this is that the addresses of memory locations are usually stored within the control structure and are not part of a gadget[4]. Consequently, variable substitution can often be achieved by altering the control structure which may allow it to evade control structure signatures.

Register substitution on the other hand in general requires the substitution of gadgets and is thus effective against both control structure signatures and code signatures.

Code Transposition Code transposition makes use of additional branches to reorder the basic blocks of the malware. In the case of data-only malware a branch is essentially a gadget sequence that sets the virtual IP to a different location in the control structure. For ROP this is achieved by increasing or decreasing the SP, for instance. Consequently, the technique requires additional gadgets similar to code transposition for traditional malware which requires additional code. Due to this fact, the mechanism is capable of defeating control structure signatures as well as code signatures.

In summary, encryption, polymorphism, and metamorphism for data-only malware can be achieved by modifying the control structure. When a modification requires additional or different gadgets, it will in addition to the control structure alter the executed code of the malware. Consequently, such modifications can be used against both control structure signatures and code signatures. From the techniques covered above this includes junk code insertion, instruction replacement, register substitution, and code transposition. However, to use these techniques, the target system must provide the necessary additional gadgets. This limitation does not apply to mechanisms that only modify the control structure, but do not change the instructions that will effectively be executed. Namely, these techniques are instruction replacement, instruction permutation, and variable substitution. While these techniques are less restrictive and thus easier to

[4]Finding gadgets that use specific addresses is often difficult. This is why one in general tries to find gadgets that operate on registers. Immediates (addresses) can then be loaded from the control structure into a register when required. In the case of ROP this is achieved with the help of a `pop <reg>; ret` gadget, which frequently occur in code.

apply, they in general can only defeat control structure signatures. The exception to the rule is instruction permutation, which does not alter the underlying instructions, but the order in which they are executed. Therefore the technique is also able to defeat code signatures.

3.6.5 Summary

In this section we continued our comparison of data-only malware and traditional malware by considering the advanced properties introduced in Section 2.1.4. We began by addressing the question whether data-only malware has the same computational ability as traditional malware. In the process we established that data-only malware can in many cases make use of a Turing complete gadget set, which enables it to perform arbitrary computations. Since the capabilities of data-only malware in the end, however, always depend on the host application whose instructions it abuses, the malware form is more restricted in its application than traditional malware.

In the next step we analyzed the level of stealth that data-only malware provides. In this case data-only malware proved superior compared to traditional malware as data-only malware does not change *any* constant system resources, while traditional malware is forced to modify the codebase.

From stealth we moved on to compare the environment dependencies of both malware forms. While data-only malware and traditional malware share the exact same dependencies on hardware, operating system, and network, we found that data-only malware in contrast to traditional malware heavily depends on the application it infects and the file format or the vulnerability it uses for infection. In addition, we discovered that data-only malware face two additional dependencies that do not exist for traditional malware: control structure dependency and switching sequence dependency. Consequently, data-only malware has more dependencies than traditional malware.

Finally, we took a look at encryption, polymorphism, and metamorphism and investigated the issue whether these techniques can similarly be applied to data-only malware in order to evade signature-based detection. During our discussion we found that all techniques can similarly applied to data-only malware. Thus data-only malware is capable of evading signature-based detection as easily as traditional malware.

3.7 Related Work

Data-only Malware To the best of our knowledge, there currently exist three other works in the field of data-only malware ([24, 61, 62]). All of them are primarily concerned with one shot data-only malware. In particular, Hund, Holz, and Freiling [62] present a non-persistent ROP-based rootkit, while Chen et al. [24] present a non-persistent JOP-based rootkit. Both rootkits essentially provide the same functionality: The infection is conducted using a hand-crafted vulnerability that is introduced into the kernel by loading a self-written kernel module. Consequently, both rootkits rely on a vulnerability-based infection mechanism. Once the vulnerability has been exploited, the first stage of the exploits will load a single data-only payload into kernel space and executed it. The payload in this particular case consists of a data-only program that is capable of hiding a process within the system by modifying the internal process list managed by the kernel. Essentially this is achieved by removing the target process from the process list. After the execution of the payload, the rootkits will then restore a valid kernel execution path to resume normal system execution.

Besides our own work, the only other successful attempt made at the creation of persistent data-only malware was performed by Hund [61], who implemented a persistent rootkit. The approach described by Hund, however, differs significantly from the techniques presented within this thesis. First of all, while Hund also identified the problem of interrupt-induced overwrites, he never found a reliable solution to cope with them. In his final solution he uses the `cli` instruction to disable interrupts, acknowledges, however, that this approach could still suffer from race conditions in practice.

Second, the switching mechanism that Hund uses and that we discussed in section 3.3.3.1 is limited in several ways. Most notably, it seems to be unsuited for the infection of 64-bit kernels as gadgets that move a kernel address into `rsp` are seldom. In fact, we did not find a single gadget for this purpose in the entire Linux 3.8 kernel binary, which has the considerable size of 18 Mb and enabled us to implement an entire rootkit as we will describe in Section 3.5.3. In addition, even if such a gadget would exist, the mechanism would still be unreliable as the OS could potentially overwrite the data-only malware by using the fixed memory area.

Third and finally, the rootkit implemented by Hund seems only to work in simple hooking scenarios due to architectural constraints. The problem arises because Hund assumes that the rootkit will be executed with interrupts disabled. Consequently, there only exists a single control structure in memory. Many functions, especially within the kernel, will, however, enable interrupts as soon as they are invoked or are unable to execute as long as interrupts are disabled. As a result, data-only malware designed according to the approach proposed by Hund will only be able to invoke a small subset of functions which severely limits the approach in practice.

Weird Machines. Shapiro, Bratus, and Smith [140] showed that it is possible to create a Turing complete language based on ELF metadata which allows one to perform arbitrary computations with the Linux loader. While our work shares similar ideas, the problems that we try to solve are fundamentally different. Shapiro, Bratus, and Smith abuse ELF metadata as input for a “weird machine” that will perform computations with the help of the Linux loader. In contrast to this, our work only uses the *intended* functionality of the loader to create a data-only program in memory. The execution of the data-only program is thereby not conducted by the loader, but by diverting the control flow of the infected binary.

3.8 Summary

In this chapter we determined the capabilities and limitations of data-only malware by conducting a detailed and systematic analysis of this novel threat and its properties. We began by explaining the fundamental concepts that form the basis of the malware form. In the process, we established that data-only malware is essentially a malicious data-only program written in a code reuse language. As a consequence, data-only malware always requires a host program, whose instructions it can reuse, to function.

Having provided the basis for our analysis, we covered the three data-only malware types that exist (one shot, persistent, and resident data-only malware) by detailing the challenges associated with their creation and explaining how one may overcome them. Most importantly, we showed that data-only malware can – against popular belief – achieve the properties of persistence and residence. As a result, data-only malware can reach the same level of sophistication as traditional malware. We demonstrated this by presenting implementations of sophisticated data-only malware capable of infecting current systems in spite of the variety of protection mechanisms that they leverage.

Finally, we concluded our analysis by performing an in-depth comparison between traditional malware and data-only malware. We found that data-only malware can, similarly to traditional malware, rely on encryption, polymorphism, and metamorphism to evade detection and even surpasses its traditional counterpart in its level of stealth. However, what is most surprising is that data-only malware can in many cases perform arbitrary computations in spite of the fact that it relies on code reuse and can thus achieve the same computational ability as traditional malware. This makes data-only malware a realistic and powerful threat that is well suited for the creation of sophisticated attacks. However, our in-depth comparison also allowed us to identify three inherit weaknesses of the malware form: it requires a host application as basis, a switching sequence for its activation, and a control structure for its execution. As we will show in Chapter 7 these dependencies are very well suited as a basis for detection mechanisms.

Chapter 4

Existing Defenses

By studying data-only malware in theory as well as in practice, we have established that this malware form represents a realistic and powerful threat. So far, however, we have only considered current protection mechanisms marginally and, with the exception of signature-based detection, entirely ignored detection mechanisms in our analysis. This leads to the question whether an existing approach can already counteract this novel threat or if data-only malware poses an immediate danger to current systems. To address this question, we will in this chapter investigate the effectiveness of existing defense mechanisms against data-only malware. In the process, we will show that data-only malware can evade all of the existing defense mechanisms except hook-based detection.

Chapter Outline. We begin by investigating the applicability of current protection mechanisms to data-only malware in Section 4.1. In Section 4.2, we consider our first detection mechanism: signature-based detection. This is followed by an analysis of anomaly-based detection in Section 4.3. In Section 4.4, we consider integrity-based detection mechanisms, before we discuss hook-based detection in Section 4.5. Finally, we summarize the results of our analysis in Section 4.6.

4.1 Protection Mechanisms

In this subsection we will briefly revisit the protection mechanisms discussed in Section 2.4.1 and examine their effectiveness against data-only malware. In the process we will cover the protection mechanisms in the exact same order as they appear in Section 2.4.1 beginning with the software-based protection mechanisms.

4.1.1 Software-based Mechanisms

StackGuard. While StackGuard makes it more difficult to *load* data-only malware using a vulnerability-based infection mechanism, it can solely protect against buffer overflows targeting the saved return address on the stack. However, there exist many other attack vectors that are not covered by the approach. For example, the mechanism is in general powerless against *n-byte* writes, an exploitation technique that we will cover in more detail in Section 5.2.2. Since data-only malware is not limited to buffer overflows, but can instead use a wide range of vulnerabilities or a file-based infection mechanism to load itself, StackGuard can only marginally reduce the threat of data-only malware.

PatchGuard. Since PatchGuard is closed source and in addition highly obfuscated, it is difficult to reason about the capabilities of the protection mechanism. However, the few sources that are available about it [130, 145, 146] indicate that PatchGuard is primarily designed to protect key kernel data structures and the OS code. Consequently, PatchGuard essentially combines hook-based and code-based detection.

Since data-only malware does not modify the codebase of the kernel, the only attack surface that it provides for PatchGuard are the changes it conducts to kernel data structures (e.g. to install hooks or to facilitate the switch to a persistent chain). To test this capability we simulated the changes that our persistent rootkit[1] conducts to kernel data structures on a fully patched Windows 7 SP1 64-bit system. While the modification of the sysenter MSRs was not detected by PatchGuard, the changes made to the system call table[2] were detected. In conclusion, PatchGuard can detect kernel level data-only malware if it installs hooks into key data structures within the system. We will defer a more detailed discussion of hook-based detection to Section 4.5. An overview of some of the structures that are protected by PatchGuard can be found in [130].

ASLR. ASLR randomizes the base address of libraries and binaries when they are loaded. Among the protection mechanisms discussed so far, ASLR is certainly the one that is most effective against data-only malware. If the attacker is unable to determine the addresses of the gadgets she requires, the attack will obviously fail. However, the mechanism has two crucial weaknesses. First, it cannot protect against any attack that abuses the loader as demonstrated in Section 3.5.2, because the loader *must* be able to resolve addresses to provide functionality such as symbol resolution. Consequently, ASLR is unable to stop file-based data-only malware.

Second, a single memory disclosure vulnerability is in general sufficient to bypass ASLR. If the attacker manages to obtain a single address, she can calculate the addresses of all other gadgets. This is due to the fact that ASLR in general only randomizes the base address of a code region, while the relative offsets between instructions remain the same. However, Snow et al. [148] have shown that even a more fine-grained ASLR implementation would not solve this issue, since an attacker could repeatedly reuse the same memory disclosure vulnerability to obtain an arbitrary number of addresses.

[1] See Section 3.5.3 for details.

[2] The Windows equivalent of the Linux system call table is the System Service Dispatch Table (SSDT) [130].

Consequently, memory disclosure vulnerabilities constitute an inherent problem of ASLR. Since research has shown that attackers generally have such vulnerabilities at their disposal when performing an attack [135, 148], ASLR can, although it seems effective in theory, often be bypassed in practice.

4.1.2 Hardware-based Mechanisms

W $\oplus$ X. Due to W $\oplus$ X memory regions are either executable *or* writable. While this approach effectively protects systems against traditional shellcode-based attacks, it is powerless against data-only malware as it does neither introduce new instructions into the system nor modify existing instructions.

SMEP. A common technique leveraged by attackers in the case of kernel exploitation is to place their shellcode into user space and to trigger the execution of the code using a kernel vulnerability [113]. This approach has the crucial advantage that an attacker can execute arbitrary code at the highest privilege level without having to find a way to load code into the kernel's memory space. SMEP hinders such attacks by ensuring that no code is executed from a user space page while operating at the highest privilege level. Since data-only malware does not leverage such an approach, however, the mechanism is unable to reduce the attack surface in this case.

SMAP. SMAP is essentially the counterpart to SMEP for data. When enabled, the CPU will generate an exception whenever a user space page is accessed while operating at the highest privilege level. As a result, data-only malware can, similar to shellcode, no longer be stored in user space and then executed from kernel space. While this minimizes the attack surface, data-only malware can still execute at the privilege level it is stored in (i.e. within user space or kernel space). In addition, the kernel must access user space data from time to time to be able to fulfill its purpose. For instance, to execute a system call, the kernel often requires user space arguments such as the address of the buffer that stores the result of the system call. Consequently, SMAP must in contrast to SMEP be temporarily disabled during normal operation. This provides an attacker with various possibilities to circumvent the mechanism. For instance, could she leverage a hook to redirect the execution flow precisely at the point in time when SMAP has been disabled.

Code Signing. In case of code signing, the integrity of a binary is verified while it is loaded. Given that a code signing approach validates the entire binary and not just its code regions, code signing can, in some cases, hinder file-based data-only malware. Since code signing faces similar restrictions as file-integrity checking, however, we will defer a more detailed discussion about the effectiveness of the mechanism to Section 4.4.1.

4.2 Signature-based Detection

In principle, signature-based detection can be used to detect data-only malware as well. The necessary signatures can thereby be created in two different ways: one can either create signatures based on the gadgets that the malware executes (code signature) or one can focus on the control structure of the malware (control structure signatures). As described in Section 3.6.4, however, data-only malware is capable of similar evasion techniques (e.g. polymorphism and metamorphism) as traditional malware. Consequently, both signature types essentially face the same limitations as traditional signatures. The most severe being that signature-based detection is vulnerable to evasion attacks and that signature generation mostly remains a time consuming manual process.

While comparing both signature types, we found that code signatures are in general more difficult to evade than control structure signatures. Thus if a signature-based detection approach is used to hinder data-only malware, code signatures should be preferred. To make evasion more difficult, detection mechanisms can, as in the case of traditional malware, employ emulation or semantic signatures. Since even anti-malware vendors acknowledge that current signature-based approaches are only able to detect 45% of all malware infections [132], however, it is highly unlikely that signature-based detection will be more successful in mitigating the threat of data-only malware as it currently is in stopping traditional malware.

4.3 Anomaly-based Detection

A very important property of anomaly-based detection with regard to data-only malware is that it focuses on the behavior of malware rather than its implementation. This enables anomaly-based detection to counteract one of most dangerous features of data-only malware. In particular, data-only malware constitutes a challenge for existing defense mechanisms since it solely consists of data. At its heart, however, data-only malware only represents a novel way to *implement* malware, while the behavior of the malware remains unchanged. As a consequence, anomaly-based detection can directly be leveraged to detect data-only malware.

That being said, anomaly-based detection, however, also faces the same limitations as in the case of traditional malware detection. Most importantly, it is usually quite difficult to identify features that are suitable for the detection of abnormal behavior and anomaly-based detection typically suffers from a high false positive rate [63]. The latter is thereby a particular severe problem, since even a small number of false positives can render a mechanism unsuitable for real world applications. Since anomaly-based detection, like signature-based detection, has so far been unable to substantially hinder traditional malware, it is implausible that current approaches will be more effective against data-only malware.

4.4 Integrity-based Detection

Having covered signature-based detection and anomaly-based detection, we will in this section discuss the effectiveness of the integrity-based detection mechanisms that we introduced in Section 2.4.2.4 against the threat of data-only malware. As before, we will address the individual mechanisms in the same order as they appear in Section 2.4.2.4.

4.4.1 File Integrity Checking

File integrity checkers can in theory prevent the execution of file-based data-only malware by checking the integrity of every file before its execution. While this approach seems simple at first glance, it is not as straightforward to realize in practice as one might think. First of all, to actually perform the comparison we require a cryptographic checksum of every file. Naturally, however, there exist many binaries for which such a checksum is not provided by default. For instance, files received via e-mail rarely come with a checksum.

To solve this problem, systems such as Tripwire [73] and Open Source SECurity (OSSEC)[3] allow their users to calculate the cryptographic checksum of a file in the current system environment. That is, they assume that the current system environment is secure and can thus be used to calculate the initial checksum that will later on be used for the comparison. There is, however, no guarantee that an attacker has not already compromised the system when the calculation is performed.

Second, to provide a reliable comparison, the benign checksums must be stored in a secure location that cannot be compromised [73]. Otherwise, an attacker could simply change a file and then update its checksum in the database used for comparison. In practice, this implies that we must have the ability to verify the integrity of the database as well (e.g. using certificates). In addition, we need to ensure that the component performing the integrity verification is in fact integer and has not been modified. This leads to a complex architecture that must perform a lot more tasks than simple file integrity checking. Instead, it must ensure the integrity of the database and the verification component (and any other component involved), which may reside within a potentially malicious environment.

Third and finally, not all files are constant. Text files, for instance, may obviously change during their lifetime. Since many word processors support script languages in the meantime, text files can thereby provide similar powerful attack vectors as executable files. Beyond that, an attacker could always resort to exploitation to infect a system with malware. Even simple text files can become security relevant if they can trigger vulnerabilities in the programs processing them. Consequently, to protect against data-only malware, one must actually not only verify constant files, but also dynamic files. Doing so reliably and efficiently, however, remains, to the best of our knowledge, an open

[3]http://www.ossec.net/

research problem. Due to this problem file integrity checkers similar to anomaly-based detection systems often suffer from false positives [157].

In conclusion, file integrity checking is capable of preventing file-based data-only malware, given that the infected file is immutable and that a cryptographic checksum for the file exists. It is thereby essential that the checksum does not only cover the code regions of the file, but also all of its data regions. Creating a secure architecture for file integrity checking that provides a checksum for every binary is difficult though. Further, validating mutable files remains an open research problem. It is probably for these reasons that file integrity checkers are only found seldom on systems today. Finally, file integrity checking cannot protect against vulnerability-based data-only malware.

4.4.2 Code Integrity Checking

Code integrity checkers have been proven to be very effective against traditional malware, even against highly sophisticated malware such as kernel level rootkits [128]. As we discussed in Section 2.4.2.4, the primary reason for this success lies in the fact that traditional malware *must* change the codebase of a system to be able to execute. While code integrity checking is predestined for the detection of traditional malware, the approach is, however, obviously unable to detect data-only malware. In fact, data-only malware was originally designed to evade this detection mechanism [62], which highlights the importance of code integrity checking for traditional malware detection.

4.4.3 Control Flow Integrity (CFI) Checking

The goal of CFI is to validate the control transfers conducted by an application using its control flow graph (CFG). While powerful in theory, CFI faces two major limitations in practice. First of all, every check of a control transfer comes at a cost [156]. The more instructions we verify, the higher the overhead. Ideally, however, we want to validate *every* control transfer of an application in order to detect data-only malware reliably. Since even a small overhead can have a huge impact for applications that are optimized for performance such as an OS kernel, it is therefore essential to reduce the runtime overhead of CFI as far as possible.

A promising approach in this direction has been presented by Bletsch et al. [12] in the form of *control flow locking (CFL)*. The central idea thereby is to validate control transfers *after* they occur in a *lazy* manner. This is accomplished with the help of *control flow locks*: whenever a branch is taken its control flow lock is set. The lock is only removed when a valid destination for the branch in the control flow is reached. A deviation of the control flow can then be detected when the lock is still set and a branch instruction is encountered, as the control flow in this case never reached a legal destination that removed the lock. While this implies that an attacker can deviate *once* from the CFG before she is detected, this lazy validation approach enables CFL to provide a better performance than traditional CFI approaches. In the experiments conducted by the authors, the maximal overhead incurred by the approach was 21%. A significant improvement considering that the original CFI mechanism proposed by Abadi et al. [1] has a maximal overhead of 45% when performing the exact same experiments. However, even in the case of CFL the overhead certainly remains substantial.

Second, the effectiveness of CFI heavily depends on the quality of the CFG used [37], which can be obtained through "source-code analysis, binary analysis, or execution profiling" [2]. For instance, to validate an indirect branch instruction, we require *exactly* the targets that the branch could transfer execution to. However, determining this target set is a challenging task, since the problem is in general undecidable [98]. To circumvent this issue and to avoid false positives CFI mechanisms usually *overapproximate* in practice. That is, they allow a larger set of control flow targets than the real precise CFG actually would allow.

Although this may seem as a minor issue at first glance, overapproximation is actually a severe problem that almost all current CFI mechanisms share (e.g. [12, 27, 181, 185, 186]) and that can be leveraged to easily bypass current approaches in practice as various researchers have shown [20, 37, 135]. In particular, due to overapproximation, CFI mechanisms become vulnerable to *mimicry* attacks where an attacker abuses the additional control transfers that a *coarse-grained* CFG allows to perform malicious computations. This approach is possible as the overapproximated CFG allows control transfers, which could actually never occur during normal execution, and CFI cannot protect against any attack that operates "within the bounds of the allowed CFG" [2]. To demonstrate this, let us consider two overapproximations that are commonly used by current mechanisms. For instance, CFI approaches commonly allow an indirect `call` instruction to target *every* function of a program (e.g. [1]), even though the call should actually only be allowed to invoke specific functions. As a result, an attacker can invoke an *arbitrary* function of the application if she controls a single indirect `call` instruction. This is why researchers currently consider ret2libc as a general limitation of CFI mechanisms [186].

Another common check performed by current CFI mechanisms is that a `ret` instruction must always return to an instruction preceded by a `call` instruction [20]. In reality, however, a `ret` instruction should only be allowed to return to the `call` that invoked the function, not to *every* location preceded by a `call` instruction. While this overapproximated "return rule" restricts the gadgets that an attacker can leverage, it is insufficient to prevent ROP attacks. In fact, as shown by Davi et al. [37], an attacker can still perform arbitrary computations if this restriction is in place. That is, ROP still remains Turing complete in spite of this restriction.

In conclusion, while CFI certainly reduces the attack surface, it faces practical problems that must be solved before it can be leveraged as a general protection mechanism against data-only malware. Most importantly, as recent research has shown, *coarse-grained* CFI is unable to provide real security and can often easily be bypassed in practice. Consequently, more research is needed to create *fine-grained* and *faster* CFI mechanisms that provide real security with an acceptable performance.

4.4.4 Data Integrity Checking

Just as code integrity checking is well-suited for the detection of traditional malware, data integrity validation would lend itself well for the detection of data-only malware. In contrast to code, data may, however, change frequently during normal operation making data integrity checking a lot more difficult than code validation. In particular, data-integrity checking requires us to find data integrity constraints that enable us to distinguish between benign and malicious changes conducted to data. As described in Section 2.4.2.4, current approaches primarily focus on validating the integrity of the OS kernel data structures. Unfortunately, however, these systems are only able to enforce data integrity checks, but are unable to generate the rules required for the actual validation. Instead, the necessary rules must be created by human experts. Given the complexity of modern OS kernels, however, it is even for experts nearly impossible to come up with rules that are capable of validating the data regions of the kernel in their entirety reliably. This is why, some researchers such as Rutkowska even go as far as to say that current "systems are [simply] not designed to be 'verifiable'" [131]. As a result, there currently exist many loopholes that can be abused by an attacker to hide data-only malware from data integrity checkers.

In conclusion, current data integrity approaches are unable to hinder the threat of data-only malware. To be able to leverage data integrity checking for malware detection, additional research in the field of automatic integrity constraint generation is required. Techniques that are able to generate signatures for kernel data structures such as the ones presented by Lin et al. [84] or Dolan-Gavitt et al. [42], could here provide a good starting for further research, since the generated signatures could potentially be used to infer integrity invariants.

4.5 Hook-based Detection

Generally speaking, malicious code requires *hooks* within the system to function. Without them, an attacker is blind to the events occurring in the system, rendering her unable to even perform simple malicious activities such as hiding of files or capturing keystrokes. Realizing this, researchers presented various hook detection mechanisms [57, 81, 174]. Although existing detection mechanisms may not yet be able to detect all hooks that are placed by malware, the remaining possibilities for malware to install hooks are constantly dwindling. Hooks that are based on code modifications are usually no longer an option, since changes to code areas are prone to code-based detection. This leaves attackers only with the option of data hooks, but even here the options are increasingly restricted by modern detection mechanisms. One of the main reasons for this is that control data in contrast to non-control data is in general static [175]. As a consequence, it is much easier to validate making hook-based detection suitable for real world applications as mechanisms such as PatchGuard show.

To illustrate the capabilities of hook-based detection at a practical example, consider the persistent rootkit we described in Section 3.5.3. The rootkit modifies the sysenter MSRs as well as function pointers within the system call table. Both of these modifications would have been detected by the hook-based detection mechanism presented by Wang et al. [175], for instance. This amply demonstrates that hook-based detection can generally be very effective in hindering data-only malware.

In conclusion, since hook-based detection mechanisms do not target the malware itself, but the hooks it installs, data-only malware is similarly prone to hook detection as traditional malware.

4.6 Summary

In this chapter, we discussed the effectiveness of current protection and detection mechanisms against data-only malware. In the process, we found that most of the existing defense mechanisms are fundamentally flawed. Of the available protection mechanisms we consider ASLR to be the most promising approach. By randomizing the address space, ASLR makes it significantly more difficult for the malware to obtain the addresses of the gadgets it requires to function. However, memory disclosure vulnerabilities often enable an attacker to easily bypass ASLR and research has unfortunately shown that such vulnerabilities are quite common [135, 148]. Thus while ASLR is strong in theory, it is in its current form unable to mitigate the threat of data-only malware. We will discuss an extension to traditional ASLR that can potentially overcome its current limitations in Section 7.3.1.

The existing detection mechanisms present a similar picture: while most of them seem capable of defending against data-only malware in theory, they have striking weaknesses in practice. An overview of all approaches is shown in Table 4.1. We currently only deem hook-based detection to be effective against data-only malware. While data-only malware might manage to hide itself, its hooks represent an apparent weakness that is well suited for its detection.

Detection	Traditional Malware	Vulnerability-based DoM	File-based DoM
Signature-based Detection	✓	✓	✓
Signature-based detection can only protect against *known* malware instances. Since more and more new malware instances appear everyday, the approach is less and less able to protect systems.			
Anomaly-based Detection	✓	✓	✓
Anomaly-based detection can detect both traditional malware as well as data-only malware, because the mechanism targets the behavior of the malware instead of its implementation. The detection approach, however, also provides the same limitations as when applied to traditional malware. Since current anti-virus software is only able to detect a small portion of traditional malware, it is unlikely that current approaches are able to significantly reduce the threat of data-only malware.			
File Integrity Checking	✓	✗	✓
File integrity checking is in general only applicable to files that rarely change. Today's systems, however, contain many non-executable data files that change frequently during normal operation and similarly provide an attack vector. In addition, file integrity checking cannot protect against vulnerability-based data-only malware.			
Code Integrity Checking	✓	✗	✗
While effective, code integrity checking is a code-based method and can therefore only protect against traditional malware.			
CFI	✓	✓	✓
CFI is in theory able to reliably detect data-only malware. Its main problem lies in the fact that CFI is only as accurate as the CFG it enforces. Since current methods only enforce a coarse-grained CFG, they are easily bypassable as recent research has amply demonstrated.			
Data Integrity Checking	✓	✓	✓
Data integrity checking could be as effective against data-only malware as code integrity checking is against traditional malware. Current systems, however, are only able to enforce integrity constraints. The creation of the constraints is left to a human expert.			
Hook-based Detection	✓	✓	✓
Since hook-based detection does not target the malware itself, but the hooks it installs, the approach can be very effective against data-only malware in practice, which we illustrated based on our persistent data-only rootkit.			

Table 4.1: Overview of existing detection mechanisms and their applicability to data-only malware (DoM). [✓=yes, ✗=no]

Chapter 5

Dynamic Hooks

Data-only malware is a dangerous threat that is capable of evading code-based detection approaches. However, besides its instructions, current malware has another Achilles' heel that similarly affects data-only malware: in general, malware must intercept events within the system to be able to fulfill its purpose [116, 175]. Event interception, however, requires malware to divert the *control flow* of the infected system at runtime. To achieve this, malware must install *hooks* in the system that facilitate the required control flow transfer on behalf of the malware whenever the desired event occurs. While sophisticated malware such as data-only malware might manage to hide itself, these hooks represent an abnormality that will be *permanently* visible within the system. This insight led to development of a wide range of protection mechanisms that do not target the malware itself, but rather detect malware infections based on the hooks that the malware introduces [57, 81, 174]. In spite of this fact, hooking techniques have in contrast to malware, where one can observe a constant evolution of techniques and mechanisms used (such as data-only malware), not changed significantly over the course of recent years. Consequently, while data-only malware may be difficult to detect, its hooks remain – as in the case of traditional malware – a fundamental weakness.

This raises the question whether hook-based detection mechanisms could be sufficient to mitigate the threat of data-only malware. To address this issue, we once more utilize a proactive approach in this chapter and investigate the security of hook-based detection mechanisms in more detail. In the process, we find that current hook-based detection mechanisms make the incorrect assumption that attackers can only abuse *persistent* control data for hooking. As a result, existing mechanisms can be circumvented by targeting *transient* control data such as return addresses instead. To illustrate this, we propose a new hooking concept, called *dynamic hooking*, that exploits vulnerabilities to dynamically change the control flow of the system at runtime. The hook itself will thereby reside within *non-control data* and remains hidden until it is triggered. As a result, there is no *evident* connection between the hook and the actual control flow change, which enables dynamic hooks to successfully evade existing detection mechanisms.

Chapter Outline. We begin by explaining why current hook defense mechanisms are vulnerable to evasion attacks in Section 5.1. With this background in mind, we discuss the concept behind our novel hooking approach in Section 5.2. In the process, we provide a high-level overview of our approach, discuss the vulnerabilities that can be exploited to realize dynamic hooks, cover the different types of dynamic hooks that exist, and explain how suitable paths for dynamic hooks can be extracted automatically. Having presented our concept, we prove the practicability of our approach by discussing the experiments that we conducted and describing detailed POC implementations of dynamic hooks for recent Linux and Windows kernels in Section 5.3. In Section 5.4, we discuss the types of transient control data that can be utilized to implement dynamic hooks and state the limitations of the proposed concept. Finally, we cover related work in Section 5.5 and summarize the chapter in Section 5.6.

5.1 The Problem with Current Hook Defenses

Although researchers proposed various systems that aim to protect control data within an application[1] [19, 81, 116, 174], the main focus of these systems lies in the protection of function pointers that are allocated on the heap or reside within the data region of the application. *Transient* control data on the other side is generally ignored by these approaches or they merely consider the protection of return addresses, which is not the only kind of transient control data as we will discuss in Section 5.4.1.

While researchers acknowledge that malware could potentially also target transient control data to modify the control flow [81, 116, 174], these attacks are usually only considered in the context of exploitation, but are not considered to be relevant for hooking. The reasoning behind this assumption is that malware generally needs to change the control flow of the target application indefinitely in order to be continuously able to intercept events. However, to *permanently* redirect the control flow, malware must target *persistent* control data as transient control data is, by definition, only used by the system for a limited amount of time. In the following, we demonstrate that this assumption is, just like the assumption that malware has to change the codebase of a system to function, false and can be used to circumvent existing defense mechanisms against hooking.

5.2 Approach

In the following, we introduce our novel hooking concept that we refer to as *dynamic hooking*. For this purpose, we first provide an overview of the concept, before we discuss the vulnerabilities that can be used to implement dynamic hooks and cover the types of dynamic hooks that exist and their properties.

[1]For more detailed information about hook-based detection we refer the reader to Section 2.4.2.5.

5.2.1 High-Level Overview

The main problem with existing hooking mechanisms is that they require the *permanent* change of code or function pointers. Consequently, the desired control flow change of the malware is *permanently* evident within the system [116]. The fundamental idea behind dynamic hooks is to solve this problem by hiding the desired control flow change within *non-control data* such that there is no clear connection between the changes that the malware conducts and the actual control flow change. This is accomplished with the help of *exploitation techniques*.

To exploit a vulnerable application, an attacker makes use of specially crafted input data that – when processed by the application – will eventually trigger a vulnerability. If the vulnerability enables the attacker to overwrite important control structures such as a return address, she will be able to modify and often control the execution flow of the application using techniques such as ROP [138].

With dynamic hooks, we apply the same concepts that are used in traditional exploitation scenarios to hooking. That is, we manipulate the input data of the functions we want to hook in such a way that we will trigger a control flow modifying vulnerability when the data is used. This effectively allows us to overwrite control data (e.g., a return address) at runtime and enables us to control the execution flow of the application similar to a traditional hook. The main difference, however, is that such a dynamic hook will reside somewhere within the data structures of the application unnoticed until its malicious payload is eventually used by the target function.

For this approach to work, we need to identify a control flow modifying vulnerability in every function that we want to hook. At first glance this seems unlikely. After all the exploitation of vulnerabilities became more and more difficult over the course of the last years due to widely deployed protection mechanisms and even though the vulnerability databases list new vulnerabilities every month these in general only affect a small number of functions. However, there is a key difference between the exploitation of traditional vulnerabilities and vulnerabilities that are used to realize dynamic hooks: the attacker *already* controls the application at the time she installs a hook. In a traditional exploit, the attacker's goal is to *gain* control over an application. To achieve this, she needs to find an input to the application that will trigger a vulnerability. That is, the attacker can only control the *external* data which is provided to the application. In the case of a dynamic hook, however, this restriction does not apply. As the attacker controls the application, she is free to access and modify *any* internal data structure of the application. This results in a much stronger attacker model when compared to traditional exploitation.

Finding and exploiting vulnerabilities in such a scenario becomes much easier for several reasons. First, many existing protection mechanisms such as ASLR, stack canaries, or $W \oplus X$ only protect against an *external* attacker, but can be easily circumvented by an attacker that controls the application. Second, the attacker can *prepare* the code (or ROP chain) she wants to execute when the vulnerability is triggered beforehand and does not have to provide it during the exploitation process. This enables the attacker to exploit vulnerabilities where traditional methods would fail due to the space constraints of the vulnerability. Third, the attack surface for dynamic hooks is much broader. The attacker cannot only attack functions that handle user input, but can also target internal functions that cannot be influenced by the user. In fact, by manipulating internal data structures, the attacker can create new vulnerabilities that would not occur during normal operation of the application, because the targeted data structures are normally only accessed and modified by the program itself. This may even allow the attacker to circumvent checks and filters within the application as the manipulated data structures may contain values that could never occur during normal operation and may thus not have been expected by the programmer. Finally, to hook a specific event, the hook may be placed anywhere within the control flow of the handling code. Consequently, the attacker must generally not find a vulnerability within a single function, but instead has a larger codebase to work with.

Example. To illustrate the concept of dynamic hooks at a concrete example, consider the following code from the `list_del` function in the Linux kernel (version 3.8):

Listing 5.1: The `list_del` function of the Linux kernel 3.8.

```
1  struct list_head {
2      struct list_head *next;
3      struct list_head *prev;
4  };
5
6  static void list_del(struct list_head *entry)
7  {
8          entry->next->prev = entry->prev;
9          entry->prev->next = entry->next;
10 }
```

This function essentially removes the given entry from its list. If the attacker controls the `next` and the `prev` field from the entry to be deleted, she essentially can trigger an arbitrary 8-byte write on a 64-bit architecture. In particular, she can write the value of `prev` into the memory address `[next + 8]` (Line 7) and the value of `next` into the memory address `[prev]` (Line 8). To use this code fragment for a dynamic hook, the attacker could, for instance, modify a specific entry within the system and set its `prev` pointer to point to the return address of the `list_del` function and its `next` pointer to point to attacker-controlled code. When the entry is deleted, the `list_del` function will then, while processing the malicious pointers, overwrite its own return address and activate the code of the attacker on its return.

The example code above was selected as the `list_del` function is used throughout the Linux kernel and demonstrates the arguments stated above. In general, this function is not exploitable by an external attacker, as the entries that are used by the function are created by other internal functions within the kernel. While these functions initialize the values of the pointers correctly, an attacker that controls the kernel can modify them arbitrarily, thus creating a new vulnerability. The `list_del` function does not expect the manipulated values and uses them without checks. This enables an attacker to conduct an arbitrary 8-byte write, which is not enough to introduce shellcode into the system, but is sufficient to transfer the control flow to a previously prepared code region. In addition, the attacker is not hindered by any of the protection mechanisms used by the Linux kernel, since she can disable $W \oplus X$ for her code[2], must not overwrite the stack canary, and knows the address of her code or can calculate the address of the location of the return address[3]. Finally, since the `list_del` function is invoked by many other functions within the kernel, a dynamic hook within this function is very effective.

5.2.2 Suited Vulnerabilities

In principle, any kind of vulnerability can be used to implement a dynamic hook. In this thesis, however, we will, for the sake of simplicity, focus on *n-byte* writes, sometimes also referred to as *write-what-where* primitives, such as the one presented in the previous example. But much of what we present can similarly be applied to other vulnerabilities such as classical buffer overflows.

N-byte writes enable an attacker to modify n bytes at an arbitrary memory location. In our example, the attacker controls an *8*-byte write to an arbitrary memory address. In x86-assembly, n-byte writes are essentially a memory `mov` instruction for which the source and the destination operand can be controlled by an attacker. An example of a potential 8-byte write vulnerability in Intel assembly syntax is the following instruction:

Listing 5.2: Example of an 8-byte write.

```
    mov [rax], rbx
```

If the attacker can control the contents of `rax` and `rbx` at the time the instruction is executed, she can misuse it for a dynamic hook. It goes without saying that such instructions appear frequently within software. In the Linux 3.8 kernel binary, for instance, we found more than 103,000 `mov` instructions similar to the one shown above that can potentially be abused for an 8-byte write. This corresponds to about 5% of all instructions (1,976,441) within the tested Linux kernel binary. Note that this does not include the approximately 58,000 one, two, or four byte write instructions. Together, this equates to a total of 8% of all instructions that are potentially vulnerable.

5.2.3 Types

Generally speaking, there are two different types of dynamic hooks: *dynamic control hooks* and *dynamic data hooks*. The former target the control flow of the victim application

[2]Note that this is necessary since the first write (Line 7) of the example will write the return address (`prev`) into the address pointed to by `[next + 8]`. While this is unproblematic for data-only malware, this will lead to write into the code section of the attacker in case of traditional malware. However, this is not an issue in practice, as the attacker can set her code to be writable and executable. In fact, this is, at the point of this writing, even the default for memory allocated in the Linux kernel.

[3]The location of the return address depends solely on the address of the kernel stack and the size of the current function's stack frame. Both values are known to the attacker as we explain in Section 5.3.2.

and can be used as an alternative to traditional hooks since they enable an attacker to intercept events within the application. Dynamic data hooks, on the other side, do not target control data, but rather other critical data structures within an application. As an example, consider that an attacker wants to install a backdoor. For this purpose, she places a dynamic hook into a control path that can be triggered from userland such as a specific system call. However, instead of changing control data, this dynamic hook will upon invocation directly overwrite the credentials of a predefined process and elevate its privileges to *root*. Since the task credentials are usually a data value, this can be achieved with a single memory write. Thus, instead of overwriting a return address, the attacker simply sets her hook to overwrite the memory location where the task credentials reside. As pointed out by Chen et al. [25], such non-control data attacks can be quite powerful.

While dynamic data hooks do not modify the control flow directly, they can be used to influence the control flow at a later point in time. Consider for instance data that resides in memory and is processed by a just-in-time compiler. If an attacker manages to overwrite this data with dynamic hooks before it gets compiled, she can influence the instructions that are introduced into the system, which can lead to arbitrary code execution [12].

5.2.4 Properties

We now cover the properties of dynamic hooks. In particular, we first discuss the individual parts of dynamic hooks, before we cover their binding to an execution path and their coverage.

Components. Dynamic hooks always consist of two integral components. On the one hand, there is the instruction that activates the hook, which we refer to as the *trigger*. In the case of a n-byte write, the trigger is the `mov` instruction that conducts the write on behalf of the attacker. Every path that leads to the execution of the trigger is referred to as a *trigger path*. On the other hand, there is the data that was manipulated by the attacker and encodes the malicious action that the attacker wants to conduct. This is the *payload* of the hook. For n-byte writes, the payload usually consists of two manipulated pointers: the *destination pointer*, which contains the address that will be written to and the *source pointer*, which specifies the value that will be written.

Binding. While the same trigger can be shared among different dynamic hooks, each hook in general requires its own payload. The reason for this is that the payload contains the actual data that specifies the control transfer. This data, however, will only be valid in a particular *context*. To overwrite a specific return address, for example, we must first be able to predict its exact location. This requires us to know the exact path leading to the use of the payload by the trigger. In practice, this means that a payload and thus the dynamic hook is usually closely *bound* to a specific execution path. The closer the connection between an execution path and a dynamic hook, the better the control of the attacker over the hook.

In an ideal situation, a dynamic hook is *exclusively* bound to a specific execution path. In this case, the payload of the hook is *only* processed in the execution path that leads to its trigger. This enables the attacker to predict possible modifications applied to the payload before its use in addition to the state of the machine at the time of the exploitation with high probability, since she must only consider a single execution path. Similar to traditional exploits, this is essential information that is required to be able to setup a dynamic hook correctly. After all, the attacker needs to correctly predict the exact address of the control data, which should be overwritten and overwrite it with the precise address of the target code region. Without knowing the exact layout of the stack as well as the transformations that may be applied to the payload before its use, this is a hard task.

If there are multiple paths that use the payload, the dynamic hook is only *loosely* bound to the path leading to the trigger instruction. The more execution paths the payload affects, the more difficult it will become for an attacker to control the hook. On the one hand, this is due to the fact that it will become increasingly difficult to predict the necessary memory addresses and transformations as has been described above. On the other hand, the more functions access the actual payload that the attacker modified, the more likely it will be that the hook introduces side effects into the application that may lead to unexpected behavior and crash the application. Consider, for instance, that an entry that is used by the `list_del` function has been modified to act as payload for a dynamic hook. If the same entry is used by a different function to iterate through all elements within the list, this will most likely lead to a crash as the `prev` and the `next` pointer do not point to the previous and next element, respectively, as would have been expected.

Coverage. Another important property of a dynamic hook is coverage: as dynamic hooks should be closely bound to the execution path containing the trigger, it is essential that this triggering path is *always* executed when the target event that should be hooked is invoked. In this case, the dynamic hook provides *full* coverage. Otherwise, the hook may only be able to intercept *some* execution paths of the target event, but not all. In that case, the hook has only *partial* coverage and must thus be combined with other dynamic hooks to be able to achieve *full* coverage of the target function. While binding is a property of the payload of the hook, coverage is a property of the trigger instruction.

5.2.5 Automated Path Extraction

So far we have discussed the concept of dynamic hooks and provided an overview of the different types of dynamic hooks and their properties. However, the creation of a dynamic hook still remains a manual process, which can, as in the case of traditional exploitation, be a very time-consuming and error-prone task especially for complex binaries such as modern OS kernels. We now describe how paths for dynamic hooks can be obtained automatically for a given binary. This is essentially a two-step process: In the first step, we make use of *static program slicing* [161, 176] to extract potential paths that could be used for a dynamic hook. In the second step, we then employ *symbolic execution* [74, 136] to verify the satisfiability of the paths and to generate detailed information for their exploitation.

5.2.5.1 Program Slicing

To find possible locations for dynamic hooks within an application, an attacker has to find triggers that make use of a payload that she can control. Since trigger instructions can be as simple as a memory move, there usually exist many triggering instructions in many paths of the application. To identify whether a particular trigger instruction can be used for a dynamic hook, it is necessary to analyze the data flow that leads to the particular instruction. One technique that can be used for this purpose is *static program slicing* [161, 176].

The basic idea behind static program slicing is to traverse back through the CFG of an application starting from a *sink* node and to extract each node that directly or indirectly influences the values used at the sink. Applied to the problem of finding dynamic hooks, static program slicing thus allows us to determine where the values of the *source* and the *destination* pointer in an n-byte write originate from. To achieve this, we first identify all potentially vulnerable `mov` instructions within a given binary. These are essentially all `mov` instructions which move a value contained within a register to a memory location specified by another register. In the next step, we then traverse the CFG of the binary backwards at the assembler level until we encounter the first instruction that modifies the source register of the move. We record this instruction and continue with our backward traversal. Instead of looking for instructions that modify the source register of the original move, however, we will from here on search for instructions that modify the source register of the last instruction we recorded. If we continue this process, we eventually obtain the register or memory location where the value that is later on contained within the source register originates. We then repeat the process for the destination register. All the instructions that we recorded using this method form a *slice* of the binary. Each slice contains all the instructions that affect a given vulnerable `mov` instruction.

We implemented a *slicer* which is capable of extracting potential paths that could be used for n-byte writes from a 64-bit Linux or Windows kernel binary. The implementation of the slicer is based on the disassembler *IDA Pro* [56]. In particular, we make use of the CFG that IDA provides to perform the above described *static interprocedural def-use analysis*. Starting from each trigger, we perform a breadth-first search in a backwards direction. We hereby make use of a register set to conduct the actual analysis. Initially, this register set consists of the source and destination register. Whenever we encounter an instruction that modifies a register included within the register set, we add the source register of the instruction to the set and remove the modified register. Since we walk backwards through the instruction stream, this effectively allows us to record and track the *def-use chains* for the source and destination register. In addition, we record all instructions that we visit along the way, in order to be able to reconstruct the path that we explored in case we consider it to be potentially exploitable.

The challenge that remains to be solved is to determine whether a slice can be used for a dynamic hook or not. To address this problem, we must know whether the registers in the vulnerable move can be controlled by an attacker. We consider this to be the case if the values of the source and destination register originate from a *global* variable. The reasoning behind this approach is that the data used within the move in this case stems from a persistent location. Consequently, to control the final `mov` instruction, an attacker can modify the pointer chain starting from the global variable.

To identify global variables in the kernel, we assume that each access to a fixed address or the *global segment register (GS)* constitutes an access to a global variable. The reason for the latter is that both the Linux and Windows kernels store important global variables that are valid for a particular CPU within a memory region pointed to by this register. For instance, both Linux and Windows store the address of the `task_struct` (`gs:0xc740`) or the `_ETHREAD` (`gs:0x188`) structure of the process that is currently executing in this memory region.

If both the source *and* the destination register originate from a fixed address or the memory region pointed to by GS, we consider the path to be potentially exploitable and record it such that it can later on be used as input for the symbolic execution engine.

5.2.5.2 Symbolic Execution

Symbolic Execution is a well-known program analysis method that has been proposed by King [74] over three decades ago. The basic idea of symbolic execution is to treat input data of interest as symbols rather than concrete values. At the beginning of the program execution each symbol can thereby represent any possible value. As we proceed through the program code, the values become constrained. Branches, for instance, set up conditions that constrain symbolic variables. Each condition can be represented as a logical formula. Based on these formulas we can then reason about the program. For example, we can determine whether a specific execution path is reachable. For this purpose we feed the logical formulas of the path in question into a SMT solver. If the path is satisfiable, it can be reached and we can obtain concrete values that satisfy the path conditions. For a more detailed introduction to symbolic execution we refer the reader to [109, 136].

We use forward symbolic execution to verify the satisfiability of our sliced paths and to produce detailed information for the creation of the dynamic hooks. In the process, we utilize the VEX IR, which is a RISC like intermediate representation with single static assignment form (SSA) properties, deeply connected to the popular Valgrind toolkit [101]. For details we refer the reader to the specification shipped with a Valgrind release.

To verify satisfiability, we transform each basic block of the sliced path into VEX IR code and execute the code symbolically. The translation to VEX IR is achieved by utilizing a python framework called *pyvex* [142]. We dismantle every VEX statement that we obtain from *pyvex* and link the components of the statements into our own data structures. These data structures are used to walk over the VEX code and by doing so, we semantically map the statements to Z3 expressions. Z3 is an open source theorem prover developed at Microsoft Research that we use to solve our formulas [95].

As we walk over the VEX code of our sliced paths, we also keep track of three global contexts, i.e., a memory context, CPU context, and the current jump condition. Each context consists of Z3 expressions that semantically mirror the current state of the execution. Additionally, each basic block also keeps track of temporary VEX IR variables in SSA. By constant propagation, we use these variables to resolve source and destination. Each store, load, and register set statement updates the corresponding context in form of Z3 expressions. Once we hit a jump condition, we ask the solver whether we can take the jump according to our context. If no solution exists, we can filter out the path. An unsatisfiable set of formulas stops execution of the current path, and we move on with the next slice.

At this point it is worth mentioning that we do not use symbolic execution in the traditional sense to achieve code coverage. Our main goal is to check whether we can walk down our paths and to determine what value sets lead us to the end of the slice. We use the symbolic formulas to generate detailed information about the controlled registers at the time the vulnerability is triggered as well as the jump conditions that must be fulfilled to actually reach the trigger. By processing over the VEX code, the solver also gives us possible values to set.

5.3 Experiments

Based on the slicer and the symbolic execution engine, we created a prototype that we used to automatically extract paths for dynamic hooks in a fully patched Windows 7 SP1 64-bit kernel and a Linux 64-bit 3.8 kernel. We chose this approach for three main reasons. First and foremost, since malware nowadays generally attacks the kernel, this approach allowed us to test the prototype in a realistic scenario. Second, kernel binaries are especially complex, which makes them well suited for a thorough test of our implementation. Finally, by targeting Windows and Linux, the experiments show that the proposed mechanism is applicable to two of the most popular OSs.

In the following, we first discuss the results that we obtained by providing detailed statistics about the automatically extracted paths for both kernels. To demonstrate how useful the prototype is when it comes to the actual creation of the hooks, we also describe two concrete POC implementations for dynamic hooks that we created based on the information that our prototype provided.

OS	*Size*	*Instructions*	*8-byte moves*	*Slices*	*Paths*
`vmlinux`	18,8 MB	1,976,441	42,130 (2.1%)	1753 (4%)	**566 (32%)**
`ntoskrnl.exe`	5,3 MB	1,330,791	26,694 (2.0%)	5450 (20%)	**379 (07%)**

Table 5.1: Overview of the 8-byte moves, the potentially exploitable slices, and the exploitable paths according to the symbolic execution engine for the analyzed Linux 3.8 64-bit (`vmlinux`) and Windows 7 SP1 64-bit (`ntoskrnl.exe`) kernels.

5.3.1 Automated Path Extraction

As stated above, we tested our prototype with a fully patched Windows 7 SP1 64-bit kernel and a Linux 64-bit 3.8 kernel. The goal of the experiment was to automatically extract trigger paths that could then either be used by a human expert to manually design dynamic hooks or to automatically generate exploits. Table 5.1 provides an overview of the obtained results.

At first, we determined the number of instructions contained within both kernel binaries for reference. In the next step, we obtained the number of potentially exploitable 8-byte `mov` instructions. In the process, we only counted those `mov` instructions that move data from one general purpose register into a memory location specified by another general purpose register with the condition that the involved registers were neither `rbp` nor `rsp`. The reason for this restriction is that our prototype implementation currently does not support a memory model, meaning that we cannot track memory store and load operations in our slicer, which is why we currently ignore any path that requires this functionality. We will cover this limitation in more detail in Section 5.4.2.

As Table 5.1 shows, about 2 % of all instructions within the tested kernels are `mov` instructions that fulfill this criteria. If we would include `rbp` and `rsp` as well, the number of potentially vulnerable move instructions would even amount to 162,264 (Linux) and 110,032 (Windows) respectively. Consequently, we only consider a small subset of the potentially vulnerable 8-byte move instructions at the moment. By supporting a memory model the number of potential move instructions would increase by a factor of four.

Next, we used the slicer to extract potentially exploitable slices for each of the identified moves. In case of Linux, the slicer considered about 4% of the `mov` instructions as potentially exploitable, while on the Windows side about 20% of the `mov` instructions were marked as possibly exploitable. We assume that the significant difference between Windows and Linux stems from the fact that Linux has substantially more `mov` instructions that store or load data from memory (61,651 vs 37,272). Since the slicer does not support a memory model, it will abort whenever such a `mov` instruction is part of a def-use chain. Due to their number, this scenario is more likely to occur on Linux than on Windows.

Finally, we symbolically executed each of the obtained slices. In total, this led to 566 exploitable paths for Linux and 379 exploitable paths for Windows. The symbolic execution engine thereby produced the required value for each conditional jump within the path and detailed information about the vulnerable `mov` instruction. In particular, the output[4] specifies exactly which memory addresses must be modified in what way to pass the conditional jumps and where the source and destination values are located. This information can directly be applied to generate exploits or to manually create a dynamic hook as we will show in the next section.

5.3.2 Prototypes

We now present three concrete examples of dynamic hooks to illustrate the capabilities and properties which have been discussed throughout the paper. We created these examples based on the output provided by our prototype. The first and the third example focus on a dynamic control hook, while the second example demonstrates a dynamic data hook. To ease the understanding of the examples, all hooks leverage a trigger instruction within the `list_del` function (as explained in Section 5.2.1) or its Windows equivalent. The first two hooks were implemented for Linux 3.8 and an Intel Core i7-2600 3.4 GHz CPU. To demonstrate that the proposed concept is similarly applicable to Windows, the third hook was implemented on a fully patched version of Windows 7 SP1 running on the same CPU.

5.3.2.1 Dynamic Control Hook: Intercepting Syscalls

A common functionality that kernel level malware requires is the possibility to intercept system calls. In this example, we show how a single dynamic hook can be used to intercept *all* system calls for a particular process. To achieve this, the hook is placed into the execution flow of the *system call handler*, which is, independent of the system call mechanism that is used (i.e., interrupt-based, sysenter-based, or syscall-based), invoked whenever a system call on the x86 architecture is executed. Its main purpose is to invoke the actual system call by using the system call number as an index into the system call table.

Similar to other functions within the kernel, the system call handler can be audited for debugging reasons. Auditing can be enabled or disabled within the flags field of the `thread_info` struct associated with each process. By setting the `TIF_SYSCALL_AUDIT` flag, in the `thread_info` struct of a particular process, every system call conducted by a process will also lead to the invocation of the auditing functions. In particular, the function `__audit_syscall_entry` will be executed before the invocation of a system call and the function `__audit_syscall_exit` will be executed after the system call, but before the system call handler hands control back to user space. In our POC implementation, the dynamic hook is set within the `__audit_syscall_exit` function.

When system call auditing is enabled, the `__audit_syscall_entry` function will record information about the system call such as the system call number and the arguments of

[4]An example of the output is provided in Section 5.3.2.3.

the system call within the audit context of the process for which the `TIF_SYSCALL_AUDIT` flag was set. While the `__audit_syscall_entry` function is responsible for recording this information, the purpose of the `__audit_syscall_exit` function is to reset the audit context of the task before the system call returns. In the process of resetting the audit context, the `__audit_syscall_exit` function invokes the inline function `audit_free_names`, which resets the `names_list` within the audit context. This function contains the following code fragment:

Listing 5.3: The `audit_free_names` function of our Linux test system.

```
1  static inline void audit_free_names(
2         struct audit_context *context) {
3      ...
4      list_for_each_entry_safe(n, next,
5            &context->names_list, list) {
6          list_del(&n->list);
7      }
8      ...
9  }
```

The `audit_free_names` function essentially iterates over the `names_list` of the audit context (Line 4) and deletes every entry within the list (Line 6). Consequently, if we control the `names_list`, we can control the entry that is passed to the `list_del` function, which in turn allows us to exploit its vulnerability. As the `names_list` is not modified by the `__audit_syscall_entry` function or anywhere else in the kernel (to the best of our knowledge), the attacker is free to modify it in any way she wants. That is, the `names_list` structure is exclusively bound to the execution path within the system call handler that we use for our dynamic hook.

While the `names_list` structure seems to be perfectly suited for a dynamic hook, the triggering path places additional constraints on the hook. The problem arises due to the fact that the `list_del` function is contained within a loop that iterates over all entries within the `names_list` list (Line 4). To iterate through the list, the loop will essentially follow the `next` pointer in every entry until one of them points back to the first element in the list, which is `&context->names_list`. Since we want to modify the `next` and the `prev` pointer of an entry within the list to conduct an arbitrary 8-byte write, we have to take this problem into account and assure that the list iteration will eventually terminate. To achieve this we initialize the audit context as shown in Figure 5.1.

The basic idea behind this setup is to make use of a special address, referred to as a "magic address", that is a valid memory address, but at the same time contains valid x86 instructions. Due to little-endian byte order, these valid instructions must be contained in reverse order within the address. In Figure 5.1, the instruction encoded into the address is a negative relative jump (`0xe6eb` (address) $\Rightarrow$ `0xebe6` (instruction)) that will upon execution transfer control to a trampoline[5], that then transfers control

[5]A trampoline is essentially a small code fragment that transfers control to an arbitrary address within memory. We make use of the trampoline as we can only encode a few instructions into the magic address. Once activated, the trampoline code can then call the actual function that our hook should invoke.

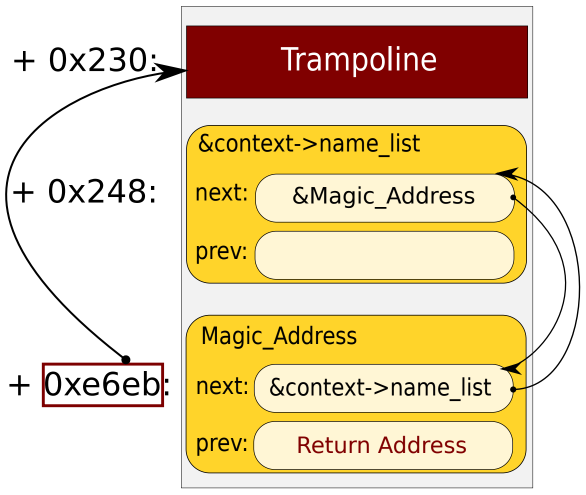


Figure 5.1: The audit context structure that the attacker uses to set a dynamic hook within `audit_free_names`. The magic address must thereby end with bytes representing valid instructions as the address of `&context->names_list` will be written into the return address, leading to the execution of the address. In this particular case we make use of a relative negative jump (`0xebe6`) to jump to our trampoline, which will then in turn call the desired function.

to an arbitrary address. Initially when the loop begins iterating over the `names_list`, it follows the `next` pointer to the first entry within the list, which is located at the magic address. The `next` pointer stored at the magic address will in turn point back to the `names_list`, thus fulfilling the loop condition. However, before the loop exits, the first entry in the list (located at the magic address) is processed by the `list_del` function. Since the `prev` pointer of this entry points to the location of a return address, the `list_del` function will overwrite this return address with the value stored in the `next` pointer ($prev \rightarrow next = next$), which points to `&context->names_list`. Consequently, as soon as the return address is used, control will be transferred to the address of `&context->names_list` where the magic address is stored, leading to the execution of the magic address and the activation of the trampoline code. Note that the hook requires the audit context region created by the attacker to be writable and executable, since the `list_del` function conducts *two* write operations as has been described in Section 5.2.1. However, this is not a big problem in practice, since every memory region allocated in the kernel is by default writable and executable.

The final problem that remains is which return address we are actually going to overwrite and how we can predict its location. As previously stated, the system call handler is invoked before every system call and it will invoke the actual function handling the syscall. Thus, if we know the stack frame size of the system call handler and the location of the kernel stack, we can predict where the return address of the function that is invoked by the syscall handler resides. The stack frame size can be obtained from the assembler code of the syscall handler, while the location of kernel stack can be obtained from a kernel variable (`get_cpu_var(kernel_stack)`). The target return address will then reside at `stack_frame_size + get_cpu_var(kernel_stack)`.

Summary. A dynamic control hook for intercepting all system calls for a particular process can be placed in the `audit_free_names` function. To ensure that execution passes through this function, we set the `TIF_SYSCALL_AUDIT` flag within the `thread_info` struct of the target process. In the next step, we modify the audit context of the target process in the way described above and use a trampoline to control the execution flow. This enables us to reliably divert the control flow at runtime. The resulting dynamic hook will have full coverage and be exclusively bound to the execution path leading to the `audit_free_names` function.

5.3.2.2 Dynamic Data Hook: Installing a Backdoor

The purpose of the second example is to demonstrate the possibilities of dynamic data hooks and to show that they are indeed a realistic threat in practice. In particular, we will make use of dynamic data hook to install a backdoor within a Linux system. The backdoor will be capable of elevating the task rights of a predefined process to root. To trigger the backdoor, the attacker must execute a `ptrace` system call on the task that contains the dynamic hook.

The `ptrace` system call is a powerful system call that enables one process to attach to another process for debugging purposes. The attached process can then control and inspect the execution of the target process. Consequently, `ptrace` is primarily used by debuggers such as gdb. In this example we are primarily interested in the `ptrace_attach` function, which tries to attach the current process to the given target process, and its counterpart the `ptrace_detach` function that essentially reverts the changes conducted by `ptrace_attach`.

To install the backdoor we will simulate that a process used the `ptrace_attach` system call to attach to the target process. This is achieved by manually applying the changes that the `ptrace_attach` function conducts to the internal data structures of the target process. Most importantly the `state` field of the task must be updated to include `__TASK_TRACED`, the `ptrace` field within the task must be set to 1, and the `parent` field must be set to the process which will later trigger the backdoor. We will defer the discussion of this last change for the moment and explain it in more detail later on.

The above described changes essentially ensure that all checks are passed and that the `ptrace_detach` function will be executed when the `ptrace` system call is invoked with the `PTRACE_DETACH` argument. Thus once the changes of the `ptrace_attach` function have been simulated, it is possible to invoke the `ptrace_detach` function on the so prepared process. The execution of the `ptrace_detach` function eventually leads to the invocation of the `__ptrace_unlink` function, which in turn invokes the `list_del` function using the `ptrace_entry` pointer within the target process as argument:

Listing 5.4: The `__ptrace_unlink` function of our Linux test system.

```
void __ptrace_unlink(
     struct task_struct *child) {
    ...
    list_del(&child->ptrace_entry);
    ...
}
```

To use this code fragment for a dynamic data hook, we modify the $ptrace\_entry \rightarrow next$ pointer and the $ptrace\_entry \rightarrow prev$ pointer of the target process. This enables us to conduct an arbitrary 8-byte write when the `list_del` function is invoked during the execution of `ptrace_detach`. In particular, we set the `prev` pointer to point to the task credentials that we want to override and the `next` pointer to an address that is writable and ends with four zero bytes. To understand this, we have to take a look at the Linux task credential structure, which defines the access rights of a process:

Listing 5.5: The task credential structure of our Linux test system.

```
struct cred {
    ...
    kuid_t uid; /* real UID */
    kgid_t gid; /* real GID */
    kuid_t suid; /* saved UID */
    kgid_t sgid; /* saved GID */
    kuid_t euid; /* effective UID */
    kgid_t egid; /* effective GID */
    ...
};
```

Each task contains three pairs of access rights and each access right pair consists of a user id and a group id. Most important for us is the effective user id (`euid`), which specifies the effective access rights of a process. Since the root user in Linux generally has the user id zero, our goal is to overwrite the `euid` field, which has a size of 4 bytes, with zeroes. If we choose an address for the `next` pointer that has its lower 32-bits set to zero and additionally set the `prev` pointer to point to the `euid` field of the process whose privileges we want to elevate, we will, due to the little endian byte order, overwrite the `euid` ($prev \rightarrow next = next$) field with zeroes and thus set the access rights of the process to root. However, because the `list_del` function will also write the `prev` pointer into the address of `[next + 8]` ($next \rightarrow prev = prev$), we have to ensure that the address used within the `next` pointer points to a writable memory region that does not contain crucial data. A possible address that can be used for this purpose is `0xffff880000000000` since this address usually points to the first 8-bytes of the physical memory of the machine, which is not used by the Linux kernel. Finally, note that we will also override the `egid` of the process with the upper 32-bits of the address in the next pointer. This will, however, not affect the process as long as it has a valid `euid`.

We can now set up a dynamic hook as follows: First, we need to select a target process that remains running on the system as it will contain the above described dynamic hook.

Good candidates are therefore background daemons such as the SSH daemon. Second, we need to specify the victim process whose privileges we want to elevate and setup the dynamic hook within the target process. Since we need to know the address of the task struct of the victim process in order to be able to set the `prev` pointer to its `euid` field, this process also needs to remain running. A good choice in this case could, for instance, be a shell process within a screen session.

To activate the backdoor, we need to call the `ptrace` syscall with the `PTRACE_DETACH` argument on the target process. However, the backdoor cannot be activated by any process because only the tracing process can detach from the traced process. Since we simulate the changes conducted by `ptrace_attach`, the process which can execute the `ptrace_detach` call, is the process that we specify as `parent` during the setup of the dynamic hook. While this ensures that the backdoor cannot be triggered by accident, this requires us to specify the process that triggers the backdoor when we setup the dynamic hook. The easiest way to solve this problem is to specify the victim process as parent of the target process. In this case the victim, whose privileges will be elevated, can trigger the backdoor itself.

Summary. A dynamic data hook can be used to implement a backdoor that can be triggered from user space with arbitrary access rights. In our example, the backdoor is closely bound to the process that was specified as the tracing process and to the execution path within `ptrace_detach`. In addition, the hook only provides partial coverage as only the detach call to a specific process will trigger it, which is desired behavior in the case of a backdoor.

5.3.2.3 Dynamic Control Hook: Process Termination

To show that the proposed hooking concept can be applied to other OSs as well, we will in our final example present a dynamic control hook that we implemented on a fully patched version of Windows 7. In particular, the hook is capable of intercepting the termination of an arbitrary process, which can, for instance, be useful in situations where a malicious process on the system is found and terminated by a security application or the user. Due to the hook, the malware would be notified of this event and could react to it.

When a process is exiting on Windows 7, the function `NtTerminateProcess` is invoked which in turn invokes various cleanup functions that prepare the termination of the process. One of these functions is `ExCleanTimerResolutionRequest`. To support a wide range of applications, Windows provides processes with the possibility to request a change to the system's clock interval [130]. This enables programs that have a demand for a faster response time to decrease the clock interval and thus to increase the number of clock-based interrupts. When a process emits such a request, the process is added to the `TimerResolutionLink` list, which is used by the OS to manage all timer resolution changes. As the name suggests, the purpose of the `ExCleanTimerResolutionRequest` function is to remove processes from the management list once they exit. Our automated path extraction tool discovered the following path within this function:

Listing 5.6: Output of our automated path extraction tool.

```
 1  ----SLICE----
 2  0x000000014042c396 mov   rax, gs:188h
 3  0x000000014042c39f mov   rbx, [rax+70h]
 4  0x000000014042c3c6 mov   rcx, [rbx+4A8h]
 5  0x000000014042c3cd mov   rax, [rbx+4B0h]
 6  0x000000014042c3d4 mov   [rax], rcx
 7  0x000000014042c3d7 mov   [rcx+8], rax
 8
 9  ----SYMBOLIC----
10  Jump Condition in: BB_0x14042c390
11  Concat(0x0, Extract(0x1f, 0x0, MEM[RBX+0x440])) >> Concat(0x0, 0xc) &1 == 0
12
13  CPU CONTEXT/CONTROLLED REGISTERS
14  RCX -> MEM[MEM[MEM[0x188+GS]+0x70]+0x4a8]
15  RAX -> MEM[MEM[MEM[0x188+GS]+0x70]+0x4b0]
```

To remove a process from the `TimerResolutionLink` list, the `ExCleanTimerResolutionRequest` function obtains the forward and the backward pointer (Line 4 and Line 5) from the `EPROCESS` structure of the process and performs the discussed list delete operation (Line 6 and Line 7). The only prerequisite for this path is that the 13th least significant bit of the memory word at location `EPROCESS`+0x440 is not set (Line 11). By manipulating this memory word and the pointers, which are located within in the `EPROCESS` struct of the process at offset 0x4A8 (Line 4) and offset 0x4B0 (line 5) respectively, we can thus perform an arbitrary 8-byte write and change the control flow. In our POC implementation, we set the forward pointer (`rcx`) to point to our shellcode and the backward pointer (`rax`) to point to the return address of `ExCleanTimerResolutionRequest`. Just as in the case of our first example, the location of the latter can be obtained by subtracting the sum of the stack frames of the invoking functions from the start address of the kernel stack, which is stored within the `InitalStack` variable contained within the `KTHREAD` structure of the thread of the process. Similarly, the area where the shellcode resides must be writable *and* executable. On Windows, we can allocate such a memory region by invoking the `ExAllocatePoolWithTag` function with the argument `NonPagedPoolExecute`.

One last problem that remains, however, is that the `TimerResolutionLink` entry structure of a process is unfortunately not exclusively bound to the path of our dynamic hook, since the `TimerResolutionLink` list is also used by other functions such as `ExpUpdateTimerResolution`. The solution to this problem is quite simple, though: since the `TimerResolutionLink` list is not critical for the execution of a process and the `ExCleanTimerResolutionRequest` function does on top of that not iterate through the list, but rather accesses the forward and backward pointers directly, we can simply remove the entry from the linked list. As a result, the manipulated entry will no longer be processed by other management functions, which will bind the `TimerResolutionLink` entry structure exclusively to our trigger path. In our experiments, removing processes from the `TimerResolutionLink` list did not affect their execution in any way. The proposed dynamic hook therefore serves as an example that an exclusive binding of a hook payload must not be given by the target application, but can also be manually enforced by the creator of the hook.

Summary. By manipulating the `TimerResolutionLink` entry structure of a process in the way described above we can install a dynamic hook and intercept the termination of an arbitrary process on Windows. While the manipulated structure is by default not exclusively bound to the trigger path, the creator of the hook can enforce an exclusive binding manually by removing the manipulated entry from its linked list. In addition, the presented dynamic hook had full coverage in our experiments. It was even triggered if we forcefully terminated the process using the task manager.

5.4 Discussion

Up to this point, we have not discussed what kinds of transient control data exist. This is why it may seem to the reader that dynamic control hooks could be mitigated by protecting return addresses alone. In this section, we cover this topic in more detail and show that this is not the case. In addition, we review the limitations of the proposed hooking concept and our current prototype.

5.4.1 Transient Control Data

Instead of targeting *persistent* control data such as function pointers in the system call table, dynamic control hooks change *transient* control data at runtime. While return addresses are a popular example of *transient* control data, it is not the only kind of transient control data that exists. For instance, if a function allocates a local function pointer, this pointer will reside on the stack and not in the data segment or the heap. Instead of overwriting the return address, an attacker can in such a case similarly target the function pointer. While this is a rather unlikely scenario, it demonstrates a very important class of attacks where a local variable on the stack is changed to achieve the desired control flow change. This class of attacks is not restricted to function pointers alone. Consider, for example, the following code from the `read`[6] system call in the Linux kernel:

Listing 5.7: The `read` system call of our Linux test system.

```
1  struct fd {
2      struct file *file;
3      int need_put;
4  };
5
6  SYSCALL_DEFINE3(read, unsigned int, fd, char
7               __user *, buf, size_t, count) {
```

[6]For better readability we directly included the `vfs_read` function into the `read` system call. In the actual code the function call in Line 13 will occur in the `vfs_read` function.

```
 8 |     struct fd f = fdget(fd);
 9 |     ...
10 |     ret = f.file->f_op->read(f.file, buf,
11 |                              count, pos);
12 |     ...
13 | }
```

In this case, a local structure (`struct fd f`) is allocated on the stack (Line 9). The structure contains a pointer to another structure (`struct file *file`), which in turn contains a function pointer that is called in Line 11. With the help of a dynamic hook, an attacker could modify the pointer within the local structure (Line 2) and point it to an attacker-controlled structure instead. If she manages this before the function call in Line 13 is executed, this will effectively allow her to control the function call and thus enable her to arbitrarily change the control flow.

Instead of targeting a return address or a function pointer directly, the attacker in this scenario modifies a local pointer on the stack. This approach enables her to control any data that the local function accesses using this pointer. In the kernel, where objects in general are accessed through pointer chains, this represents a powerful attack vector, which effectively provides control over *any* object that the pointer references. In addition, the structure that the attacker controls can itself be transient and must *not* be connected to any other structure within the kernel. This enables the attacker to evade any security mechanism that iterates through the graph of kernel objects in order to find hidden objects, which is very common in the field of VMI [19, 134], for instance.

Since the code shown above exists in many places within the Linux kernel, this example demonstrates that dynamic hooks are not necessarily restricted to return addresses to force a control flow change. Instead they can also target other transient data on the stack. This must be taken into account when one considers countermeasures against dynamic hooks.

5.4.2 Limitations

Dynamic Hooks. Dynamic hooks essentially face two limitations. First and foremost, not every function may contain a vulnerability that can be used to implement a dynamic hook. In contrast, it is likely that there are functions which are immune against the attack. However, this is not a big problem in practice: if a particular function cannot be hooked directly, it may still be possible to intercept calls to the function by hooking a function that immediately precedes or follows the function in the execution flow. After all, not every function contains a function pointer either. Nevertheless have function pointer hooks been proven to be very effective in practice.

Second, similar to traditional exploits, a dynamic hook may face restrictions that are caused by the vulnerability it is exploiting. So may specific hooks such as the one presented in our first prototype (see Section 5.3.2.1) require that certain memory areas are writable and executable. Depending on its restrictions, a dynamic hook may therefore not be suitable for every scenario. This, however, heavily depends on the particular hook.

Automated Path Extraction. While our prototype already produces very valuable paths that can be used to implement powerful dynamic hooks as we have shown in Section 5.3.2, it also faces some limitations. First, our slicer does not yet support a detailed memory model. As a result, we are unable to find dynamic hooks on paths where registers, which are currently monitored, are loaded with values from the stack. This situation frequently occurs when subfunctions are called. In this case, the calling function often stores register values temporarily on the stack to guarantee that they are not overwritten by the subfunction. During our experiments, the slicer ignored 79,853 such paths due to this restriction.

Second, the symbolic execution engine currently only handles a subset of the available x86 instructions. Most importantly, it is unable to handle some instructions that are a ring-0 privilege. This is, however, a restriction in the VEX intermediate language. In the experiments we conducted, this led to 949 (55%, Linux) and 4,908 (90%, Windows) paths that could not be verified.

Finally, the slicer and the symbolic execution engine currently do not consider the properties of *binding* and *coverage*, while determining whether a path could be used for a dynamic hook or not. Consequently, not all of the paths extracted by our prototype will be suited for the implementation of a dynamic hook. As described in Section 5.2.4, especially the property of binding can be a limiting factor. If a payload is only loosely bound, it is likely that the hook will introduce side effects that can lead to a crash of the system. Determining automatically whether a path has exclusive binding or full coverage is difficult though. As the discussed POC implementations show, even payloads that initially seem unsuited for the implementation of a dynamic hook can through subtle manipulations of the involved data structures yield very reliable hooks. To designate the binding of a payload, we thus not only have to identify whether a payload is used in multiple locations, but we also have to establish how many of those usages can be controlled by the attacker. This requires a profound semantic understanding of the data structures and functions involved.

5.5 Related Work

To the best of our knowledge, Petroni et al. [116] were the first to consider the hooking of transient control data. However, their work is primarily focused on the detection of *persistent* control flow modifications. Attacks on transient control data are thereby only mentioned as a limitation of their system. Hofmann et al. [57] presented a “return to schedule” rootkit that overwrites return addresses of sleeping processes to periodically invoke itself and evade hook detection mechanisms. While related to our work, this approach enables the rootkit only to reschedule itself, but it is not universal and does not allow the rootkit to intercept events.

In addition, there has also been a lot of work concerned with the possibilities of non-control data attacks. Chen et al. [25] were the first to demonstrate that non-control data attacks are indeed a dangerous and realistic threat. Sparks and Butler [152] presented DKOM as a general mechanism to hide objects within kernel space. Baliga et al. [8] extended this work and presented another class of stealthy attacks that do not have the goal of hiding objects, but rather target crucial kernel data structures to subvert the integrity of the system. Finally, Prakash et al. [122] discussed the manipulation of semantic values in the kernel to evade VMI.

5.6 Summary

We presented a novel hooking concept that we coined dynamic hooks. The main insight behind this concept is that existing defense mechanisms against hooking are based on the assumption that hooks can only be installed within *persistent* control data. Dynamic hooks exploit this assumption in order to break existing defenses by targeting *transient* control data instead. This is achieved by applying exploitation techniques to the problem of hooking. To install a dynamic hook, an attacker will modify the internal data structures of an application in such a way that its usage will trigger a vulnerability at *runtime*. The hook thereby only consists of the modified data as well as the exploitation logic. This results in a powerful attack model with a wide range of possibilities as the attacker can make use of the entire arsenal of exploitation mechanisms to achieve her goal. Since dynamic hooks only modify non-control data, they are considerably more difficult to detect than other forms of hooking.

To demonstrate the universal applicability of the approach, we implemented a prototype that is capable of automatically extracting paths for dynamic hooks from recent Linux and Windows kernels. The experiments that we conducted prove that dynamic hooks are not only a dangerous, but also a realistic threat that can be applied to practical scenarios such as system call hooking and backdooring.

In conclusion, dynamic hooks illustrate the significance of incorrect assumptions for the security of defense mechanisms. By exploiting this effective and powerful form of hooking, data-only malware is able to evade existing hook-based detection mechanisms.

Chapter 6

The X-TIER Framework

We have shown that data-only malware is a dangerous threat, which is, when combined with dynamic hooks, capable of evading many existing defenses including code-based detection and hook-based detection mechanisms. Since this leaves current systems mostly defenseless against this new malware form, the question arises as to how we can mitigate this novel threat. To provide an effective solution, we will address this issue systematically. In the first step, we will provide a *secure* and *flexible* foundation for countermeasures against data-only malware, to ensure the security and reliability of the countermeasures themselves. For this purpose, we will in this chapter present a general framework for the *detection* and *removal* of malware. Based on this framework we will then describe concrete countermeasures against data-only malware in the next chapter of the thesis.

A technology that comes instantly to mind when we think about malware detection is virtualization. One of the most important properties that virtualization provides is a complete and untainted view of the guest's state. Since the state of the guest encompasses all volatile and non-volatile memory, malware *must*, by definition, be contained within the state as long as it is running or stored on the victim's machine. This line of reasoning leads to the following observation: *every malware form, including data-only malware, that is residing within a VM is visible to the virtualizing hypervisor and thus by extension to all VMI-based security applications implemented on top of it.* Consequently, VMI-based security applications are predestined for the detection and removal of malware. However, to be able to leverage their full potential, one must first solve the *semantic gap* problem.

In the following, we present a secure, elegant, and universal approach for bridging the semantic gap. In particular, we introduce our framework X-TIER, which provides security applications residing on the hypervisor level with the possibility of injecting kernel modules, also referred to as drivers, from the hypervisor into a running VM. Once injected, the modules will be executed securely within the untrusted guest system. In the process, they can apply arbitrary changes to the guest and can even invoke external functions without loss of security. By providing a universal communication channel, X-TIER enables injected modules to transfer arbitrary information to the hypervisor, which effectively allows security applications to circumvent the semantic gap. Due to these properties, X-TIER is well-suited as a secure and flexible foundation for malware detection and removal as we will demonstrate based on multiple example applications.

Chapter Outline. We begin by stating the goals of our framework in Section 6.1. With this background in mind we will discuss the attacker model that we assume and the requirements of our framework, which are directly derived from our goals, in Section 6.2. From the requirements, we will move on to discuss the system design of X-TIER in Section 6.3. Once the system has been introduced, we will provide an evaluation of our framework as well as a discussion of its functionality and security in Section 6.4. In Section 6.5 we will compare our framework to related work. Finally, we will summarize the chapter in Section 6.6.

6.1 Goals

When designing a framework for malware detection and removal, it makes sense to not just consider data-only malware by itself, but to include traditional malware in our thought process as well. In fact, our framework should be general enough to function as a basis for the detection and removal of *all* malware forms. To be suitable for this purpose, it must achieve three fundamental properties:

G1 **Secure Execution Environment.** Since the primary intention behind our framework is to function as a basis for malware detection and removal, it is essential that malware is unable to attack the security applications running on top of the framework. That is, the framework should provide a *secure execution environment* for all applications based upon it. Most importantly, this environment must ensure that security applications can remain functional even if malware is present on the system they monitor. This enables our framework to provide a secure basis for malware detection as well as for malware removal. We will discuss to which extend our framework reaches this goal in Section 6.4.2.

G2 **Full State Access.** Similar important to a secure execution environment, is the ability for malware detection mechanisms to observe and access the *entire high-level* state of the system. It is thereby crucial that the view of the detection mechanism cannot be *controlled* by the malware. Otherwise malware might be able to evade detection by hiding within inaccessible or concealed system areas.

Note that while this goal is related to Goal G1, a secure execution environment does not necessarily lead to full state access. For example, a security application residing on the hypervisor level runs within a secure environment, but it does not have full access to the state of the VMs, due to the semantic gap problem.

G3 **Reliable Event Interception.** Finally, defense mechanisms must have the possibility to *intercept* events within the monitored system, which is a feature that is generally required for the *prevention* of malware infections. An anti-virus software may, for instance, want to intercept the opening of files to be able to detect malicious programs *before* they are executed. Analog to the case of accessing the state, it is thereby crucial that the event interception mechanism cannot be circumvented by the malware.

The fundamental properties that we identified and stated above almost exactly correspond to the three main features of VMI – *isolation*, *inspection*, and *interposition* – that we described in Section 2.3.2. This emphasizes the importance of VMI for malware detection and removal. However, as we stated in Section 2.3.3, VMI comes at a cost: while a security application that resides on the hypervisor level has a complete view of all guest systems, its view is limited to the binary representation of the state of each guest. This is due to the fact that the security application lacks the *semantic knowledge* of the guest OS that is necessary to interpret the state correctly. It is, however, exactly this semantic view of the guest system that we require for our framework to be useful (G2).

To provide a framework that combines strong isolation properties, with reliable event interception, and full state access, we must thus bridge the *semantic gap*. To achieve this, we can either make use of an in-band approach, an out-of-band approach, or derivation. The latter, derivation, however, only enables us to bridge a part of the semantic gap, which violates our goal of *full state access* (G2). This only leaves us with an in-band or an out-of-band approach. While out-of-band approaches are in theory able to fully reconstruct the state of a guest, this has, to the best of our knowledge, not been accomplished in practice so far. Modern kernels simply seem to complex to be able to achieve this task automatically, which is why currently even the most sophisticated approaches such as Insight [133] rely on expert knowledge to function. In spite of this, these approaches only manage to narrow the semantic gap, but cannot close it. This is why we chose to use an in-band approach for our framework.

To circumvent the semantic gap, our system enables VMI-based applications to inject kernel modules from the hypervisor into running VMs. Once injected, a module will have the possibility to access the entire high-level state of the guest without loss of security. To achieve this, the injected modules will be protected from hypervisor during their execution in the untrusted guest system. Consequently, our system effectively combines the security of an out-of-band approach with the accessibility of an in-band approach. With this background in mind, we will first state the assumptions and the requirements of our system, before we cover the design of our framework in more detail.

6.2 Assumptions & Requirements

As stated in the last section, the first goal of our framework is to provide a secure execution environment (G1). To be able to validate whether our framework can achieve this goal, we assume an attacker model where the attacker has gained *full* control over a VM and can tamper with any part of the guest OS. Consequently, it is crucial that the code injected through our system can rely on strong security properties given that it is executed in such a hostile environment. While we provide the attacker with the capability to fully control VMs, we assume, in accordance with Section 1.4, that the hypervisor and the underlying hardware are secure. This leads to the following security requirements:

S1 **Isolation.** The code injected into a VM should be isolated from the guest's code in the following way: Any code within the guest that is not *invoked* by the injected code can neither access nor modify its code or data regions. Notice that this small limitation is necessary, since it is intended behavior that invoked functions have the possibility of accessing and modifying variables of the injected code. The access of the invoked functions should, however, be restricted to the variables that the module explicitly provided to the function (e.g. as arguments). We will explain how our framework achieves this property in Section 6.3.2.2 and discuss the security of our approach in Section 6.4.2.

S2 **Error Resistance.** Faults and exceptions occurring within the injected code should be handled within the hypervisor and not within the guest. This ensures that no guest OS code is executed in the event of an error, which leaves our system in control even if the injected code is faulty.

S3 **Stealth.** To achieve a high-level of stealth (see Section 2.3.3.1 for details), the injection of code should not leave any traces within the guest unless the injected code purposefully modifies the guest's state.

S4 **Guest Independence.** The injection of a module must be achieved without the help of any functions within the guest. Otherwise an attacker could intercept injection attempts from within the VM.

These security requirements ensure that injected code will be protected by the hypervisor during its execution within an untrusted VM. However, another important goal of our framework is to bridge the semantic gap and to provide security mechanisms residing on the hypervisor level with full access to the guest's state (G2). Consequently, injected modules must be able to read and write data structures as well as to invoke functions within the guest OS. This leads to the following functional requirements:

First and foremost, the injected code needs access to the most important functions and data structures within the guest kernel's address space. A useful approach should avoid to constrain or burden the programmer in their doing. In fact, the development of injectable code should be as straightforward as writing regular code for the respective

OS. Kernel modules[1] seem to be perfectly suited to fulfill both of these requirements, since they can be developed with compiler support and enable one to access all functions and data structures within in the kernel in a familiar and intuitive way.

In addition to the possibility of executing kernel functions within a VM, it is necessary to provide a communication channel between the component within the hypervisor and the injected kernel module, such that information obtained within the VM can be transferred to the component outside of the VM. Otherwise the system will fail to bridge the semantic gap, since none of the obtained information will be accessible to the hypervisor.

Finally, the overall performance of the system should primarily depend on the runtime of the injected code, but not on the process of injecting and removing it. As a consequence, the system for code injection should be designed in a way such that the actual injection and removal mechanism is lightweight and does not lead to a significant overhead.

The following list summarizes the functional requirements described above:

F1 **Kernel Module Injection.** The system should be capable of injecting existing kernel modules into a VM. Thereby, the constraints for the creation of these kernel modules should be as few as possible.

F2 **Modification.** An injected kernel module should have the ability to apply permanent changes to the guest state. This widens the range of security applications that are supported by our system beyond simple information retrieval to sophisticated malware removal.

F3 **Communication Channel.** To be able to transfer information from the injected code to the hypervisor and vice versa, a communication channel must be provided between both components.

F4 **Performance.** The overall system design should keep the performance overhead of the injection process to a minimum, including the injection itself, the removal of the injected code, and the management tasks during runtime.

6.3 System Design

The overall design of our system is shown in Figure 6.1. As one can see, the architecture of X-TIER consists of three components: A preprocessing component, an injection component, and a communication component. In the following, each of these components will be covered in a subsection of its own.

[1]Kernel modules are a very fundamental concept that is supported by many OSs. However, the name that is used to refer to this general mechanism may vary from OS to OS. Within this thesis we will use the Linux term kernel module to refer to kernel code that can be loaded into an OS during runtime to extend its functionality.

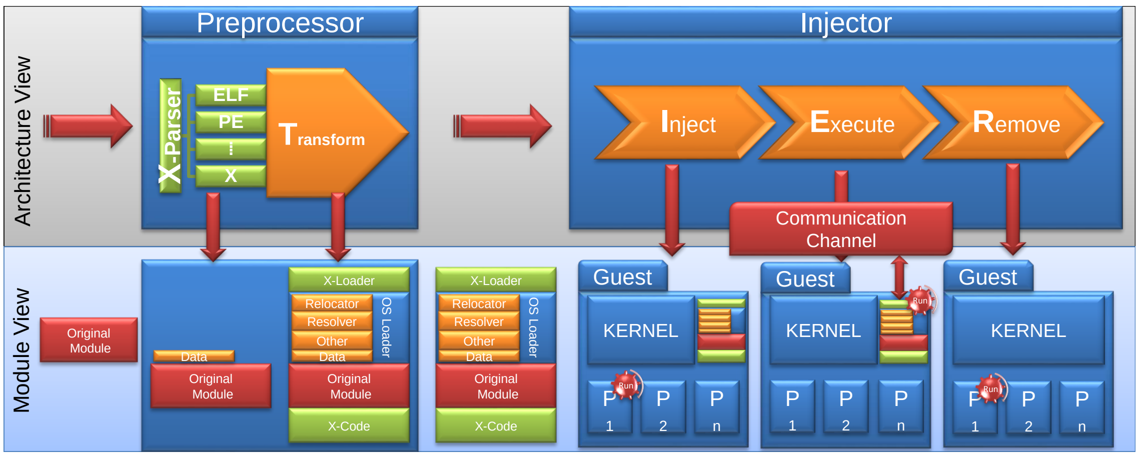


Figure 6.1: The architectural view of our system is shown in the upper-half of the picture, while the underpart shows the effects of the architectural components on a kernel module that is injected into a VM.

6.3.1 Preprocessor

To provide a general system for hypervisor-based kernel module injection, it is necessary to decouple the injection process from the format of the module that should be injected. In our system this is realized through the Preprocessor, which preprocesses each module and converts it into a custom format that we refer to as *X-Format*. By converting the different existing module formats into the X-Format, our system can operate on a single common module format. This effectively makes the *injection process* independent from the guest OS and allows our system to provide an universal mechanism for kernel code injection (F1) on the x86 architecture. In the following, we will first specify the X-Format more closely, before we describe how a module can be converted to it.

6.3.1.1 The X-Format

The idea behind the X-Format is to provide a single common structure for module injection in which existing kernel module formats such as ELF (Linux) and PE (Windows) can be embedded. This is achieved by defining a wrapper format, the X-Format, that is capable of encapsulating all of the different existing module formats. By wrapping the existing formats into a common structure, it can be guaranteed that all kernel modules provide a mutual interface and fulfill the necessary requirements for the use with our system.

The common structure of the X-Format is shown in Figure 6.2. As one can see, the X-Format consists of four main parts. At the beginning of the X-Format resides the X-Loader (1). The X-Loader functions as the common entry point for all modules in X-Format and controls the execution of the preprocessing phase of a module. Similar to normal executables, kernel modules must usually be preprocessed before they can be executed. This preprocessing is normally conducted by the OS when a kernel module is

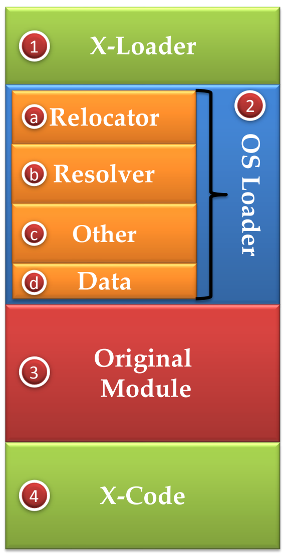


Figure 6.2: The structure of the X-Format.

loaded. Since we inject kernel modules from the outside into a VM without the support of the guest OS (S4), the necessary preprocessing steps have to be executed by our system. The individual code that is required to conduct this preprocessing phase thereby depends on the format of the module as well as the guest OS in use. To provide a mutual interface, the X-Format standardizes the loading by appending the necessary OS specific loader code (2) to the X-Loader, which will invoke it at runtime.

Although the preprocessing code varies from OS to OS, it encompasses in general at least two steps that are reflected in the design of the X-Format: Relocation (2a) and Symbol Resolution (2b). The former step, Relocation, is required to ensure that a position independent kernel module can be executed. In contrast to position dependent code, a position independent kernel module does not need be loaded to a fixed memory address in order to be able to execute. Instead, the module provides a list of addresses that have to be updated, once the memory address where a module will be loaded to has been determined. During Relocation this list is processed and the given addresses are adjusted according to the new base address of the module.

Besides Relocation, Symbol Resolution is the second common step that is usually executed by the module loader. As the name suggests, the purpose of this step is to resolve the addresses of any external kernel symbols that the module uses. How this resolution is conducted heavily depends on the OS and will not be described in detail within this thesis.

Since Relocation and Symbol Resolution may not be the only steps a specific OS executes for preprocessing, the X-Format provides a third code area (2c) for preprocessing that can contain any other code that a specific OS may require. Finally, the OS loader code section is completed by a data area (2d) that contains all of the required information for the specific loader code sections. This data includes the necessary addresses for relocation, the symbols that must be resolved, and any other data that may be required for the OS specific loader code (2c), if any. It is the task of the X-Loader to ensure that every OS loader code segment will have access to its data area, before it transfers control to the segment.

After the loader code of the X-Format, follows the kernel module (3) itself. The loader code will patch this module at runtime. Thus when the X-Loader finally transfers the control to the entry point of the module, it will be ready to execute from its current memory location. Since the X-Loader invokes the entry point of the module, control will be returned to the X-Loader once the entry point function has been executed. Thus the X-Loader code is not only the entry point of a module in X-Format, but also the exit point. This enables the X-Loader to notify our system when an injected module has finished its execution and can be removed. Notice that this event is essential for the functioning of our system.

Finally, the last part of the X-Format is the X-Code section (4). This section contains wrapper functions that are required for security purposes and the communication (F3) between the injected module and our system. We will defer the description of these wrapper functions to Section 6.3.2.2 and Section 6.3.3.2, respectively.

6.3.1.2 Conversion to X-Format

The conversion of a kernel module into X-Format is conducted in two stages. First the module is processed by the *X-Parser* component of our system as shown in Figure 6.1. The X-Parser component is the only part of our framework that must be aware of the different existing module formats. Its main responsibility is to parse the format of a given module and to extract the information needed to convert the module to the X-Format. This information consists of the entry point of the module as well as the data required for the loading stage of the module.

The Parsing Stage is followed by a common *Transformation Stage* that is equal for all module formats. During this stage the module will be converted to the common X-Format. This is achieved by adding the necessary code blocks to the original kernel module using the information that was collected by the X-Parser.

6.3.2 Injector

The process of injecting a module in X-Format from the outside into a VM consists of three individual steps that are shown in Figure 6.1. In the first step, the module must be loaded into the memory of the guest, which we refer to as the *Injection Phase*. Then the actual *Execution Phase* begins, where the control flow of the VM is altered and execution is transferred to the injected binary. This phase is particular important for the security of the proposed mechanism (G1), since it must be ensured that the injected binary is isolated and concealed during its execution within a potentially malicious guest.

Finally, the injected binary must be removed from the guest after it has finished its execution. This *Removal Phase* concludes the injection process and control should therefore be returned to the guest by resuming its normal execution. In the following, each of the above described steps will be discussed in more detail.

6.3.2.1 Injection

Before a kernel module can be executed in the context a VM, it must be loaded into the memory of the guest. For this purpose we reserve two memory regions within the guest: a memory region for the module itself and an additional memory region for the stack that will be used during its execution. Although the latter memory region is not required from a technical point of view, reserving a separate memory region for the stack allows us to achieve stealth (S3) without having to restore the complete kernel stack memory region of the VM after the execution of the injected module.

Reserving a memory region within the guest is a twofold process. First, a physical memory region has to be selected that will contain the data of the new memory region and is accessible to the guest. Next, a virtual memory mapping has to be established such that the previously selected physical memory region can be accessed by the hardware. Both steps will be described in more detail below.

Reserving Physical Memory. Instead of replacing existing data, we make use of the fact that additional guest physical memory can be allocated at runtime [54]. Using this approach has several important advantages. First of all, the guest OS will not be aware of the newly allocated memory regions, which effectively increases the isolation (S1) and stealth (S3) of the injected module. Second, the previously existing guest physical memory regions will remain untouched, which avoids unnecessary complications where data that was replaced during the injection is actually required by the injected module. Third, the approach will lead to a better performance (F4), because code that has been injected into a guest can easily be removed after execution. Instead of having to restore the physical memory region of the guest that the injected code occupied, all we have to do in this case is to free the allocated memory region.

Establishing The Memory Mapping. Once the necessary guest physical memory areas have been allocated, a virtual memory mapping has to be established that allows the hardware to access these memory areas using virtual addresses. Existing approaches (e.g. [112, 141]) that try to protect code executing within a VM from the hypervisor, often carefully create this mapping in order to provide isolation (S1). However, since our system uses runtime isolation[2] to protect the injected code during execution and does not rely on memory access flags for this purpose, the creation of the virtual address

[2]Runtime isolation will be discussed in more detail in Section 6.3.2.2.

mapping is not critical for the security of our system. Thus we only have to consider functional requirements at this point.

In order to be able to access kernel data structures and functions (G2) it must be ensured that the injected code has access to the relevant memory areas. To achieve this we have to provide the injected memory regions with the necessary access rights and have to make sure that kernel data and code regions remain mapped within the virtual address space. The first problem can be solved by setting the appropriate flags on all the segment descriptors and page table entries that are used to map the injected memory regions. To leave the existing mappings intact, we will further only modify page table entries that are free or reference user code or user data areas. In the process, it is important that we select a continuous virtual memory region to map each of the physical memory regions. Otherwise relative branch instructions contained within the injected code would no longer be functional.

Finally, all pages, even code pages, that are injected should be marked as writable, such that the loader code that is part of the X-Format is able to modify the injected code at runtime. Since, as we will explain in more detail in the next section, the injected code is the only one able to access this memory region, this is not a security issue. Instead, this allows our system to even support self-modifying code.

6.3.2.2 Execution

Once a module has been injected into a VM, control needs to be transferred to the entry point of the module to execute it. This can be achieved by manipulating the registers of the guest directly. In particular, we have to set the IP to the entry point of the injected module, which is the first byte of the X-Loader, and the SP as well as the FP to point to the newly allocated stack region. However, before these changes are applied, the value of all general purpose registers must be stored on the hypervisor level such that the original register values can be restored after the injected module finished its execution. In addition, it must be ensured that the injected code is executed at the highest privilege level (CPL 0) by setting the bits in the segment registers accordingly.

While the execution of a module can be triggered by simply setting the IP of the guest to the injected module, isolating the module from other code within the VM poses a challenge. To meet the security requirements of isolation (S1) and stealth (S3) it is necessary that the injected module cannot be accessed during runtime by other user or kernel processes within the VM. If we would allow other code within the VM to execute concurrently to the injected code, this could only be achieved by trapping all read accesses to the injected memory regions. Although this approach is possible, it would lead to a high performance overhead, which violates one of our functional requirements (F4). This is why our system makes instead use of two novel techniques to achieve this goal: *runtime isolation* and *function call unmapping*. Both of these techniques will be described in more detail below. For the sake of simplicity, we will assume a single core VM. We discuss how these techniques can be applied to multi-core systems in Section 6.4.4.

The main idea behind runtime isolation is to execute an injected module *atomically* within the guest. This requires that our system disables timer interrupts within the VM by clearing the interrupt enable flag (IF) within the RFLAGS register. Consequently, the injected module will no longer be interrupted during its execution. However, other guest OS code could still be executed in the case of an exception or an external interrupt. To avoid this problem, X-TIER further intercepts all exceptions and interrupts that occur within the VM on the hypervisor level by enabling every bit in the exception bitmap [64] and setting the interrupt descriptor table register (IDTR) base to 32 as suggested by Pfoh et al. [120], respectively. This will constrict the execution of the VM to the injected module. All other code within the VM will effectively be frozen during the runtime of the module. Even in the event of an exception or an interrupt, no guest OS code will be executed, which provides error-resistance (S2).

The only problem that remains is the handling of external function calls. If the injected module invokes an external function, this function will have access to the module's code and data regions in spite of runtime isolation. X-TIER solves this problem by temporarily removing an injected module from the guest's memory whenever an external function is invoked, which is why we coined the technique *function call unmapping*. For this purpose, the Preprocessor adds an individual wrapper for each external function that is used by a module to the X-Code section of an X-Module and additionally modifies each external function call such that it will invoke the wrapper. As a result, all external function calls within an X-Module will actually invoke wrapper functions.

Once invoked, it is the task of the wrapper to prepare the external function call. In particular, this means that the wrapper must copy all data structures that will be required by the external function and reside within the module's data area to a memory region that will be accessible to the function. During this process, the wrapper must also update any pointers that are used within the data structures such that they no longer point to the original data structures but to their copies. To accomplish this, our system makes use of a special memory area, the external function area, which is reserved by X-TIER during the module injection phase and is used as stack region for external function calls. The external function area is the only memory area that is not removed during an external function call. Since this area only contains data that must be accessible to a function (e.g. the function arguments) this is not a security issue.

After the necessary data was copied, the wrapper will modify the function arguments that were provided by the module such that every reference points to the copied data structures. Finally, it will modify the SP to point to the external function region, place the modified function arguments into the correct register and stack locations as it would do if it would invoke the external function, and use the communication channel of our framework (see next section) to notify X-TIER of the function call. In the process, the wrapper will also provide X-TIER with the address of the external function, which the wrapper in turn obtains during the symbol resolution phase of the X-Module.

Upon receiving the notification that an external function is about to be executed, X-TIER will first unmap the injected module from the VM's memory by marking all memory regions of the module as not present within the Extended Page Tables (EPT). Next, it will reenable interrupts and invoke the external function from the hypervisor by pushing the current IP on the stack and setting the IP to the specified address. This will trigger the execution of the external function within the VM. As soon as the external function returns, an EPT violation will occur, since X-TIER placed a return address on the stack that is no longer accessible. If the current IP coincides with the value that our system pushed on the stack, this event will be interpreted as the return of the external function call. In this case, our system will reenable the interception of interrupts and will return the control to the wrapper. The wrapper will then restore the stack and copy the possibly modified function arguments back from the external function area to their original location. Finally, the wrapper returns control to the X-Module which concludes the external function call.

6.3.2.3 Removal

When the injected module has finished its execution, the hypervisor component needs to be notified that the module can removed and control can be returned to the VM. In our system, this is realized through the X-Loader component. Since the X-Loader component invokes the initialization function of the injected module, control will be returned to the X-Loader component as soon as the function returns. Once the X-Loader regains control of the execution, it will notify X-TIER that the injected module has finished its execution and can be removed. This implies that all code that an injected module wants to execute must be located within the initialization function or be called by it.

Before we can return the control back to the VM, all changes that were conducted during the injection phase must be reverted to achieve stealth (S3). To ensure the VM can no longer access the injected module, all memory mappings that were created during the *Injection Phase* will be removed. In the next step, the now unmapped physical memory regions of the injected module are removed from the VM. Finally, the original values of all general purpose registers are restored. This last step reverts all changes that were conducted during the injection process and allows the VM to resume its execution from the last IP before the injection. Notice that it is not the responsibility of the injection system to remove any changes that were conducted by the injected module during the *Execution Phase*, since this would violate the functional requirement of modification (F2).

6.3.3 Communication

The system we described so far, provides us with the possibility to inject kernel modules into a VM from the hypervisor. With the help of these kernel modules we can access data structures and functions within the guest kernel, which effectively allows us to circumvent the semantic gap. However, up until now, the information that we obtain through this mechanism, is confined within the VM. What is required is a communication channel (F3) that allows us to transfer the information obtained by an injected module to the hypervisor. Within this section we will first describe the realization of this communication channel within our system, before we depict how it can be used in an intuitive way in conjunction with output functions.

6.3.3.1 Communication Channel

Our system provides a hypercall-based communication channel that allows an injected kernel module to send predefined commands to the Injector. To communicate with the Injector, an injected module will raise a predefined interrupt, which we will in the following refer to as *hypercall interrupt*. Due to runtime isolation, the invocation of the hypercall interrupt will lead to a VM Exit. Once control reaches the hypervisor level, the Injector can identify communication attempts from the module by inspecting each interrupt and looking for the occurrence of the hypercall interrupt.

To send specific commands to the Injector, the injected module will in addition to the hypercall interrupt make use of a predetermined general purpose register, which is used as *command* register. Based on the value of this register, the Injector can then upon the receipt of the hypercall interrupt determine which type of action the injected module wants to execute. For this purpose, our system supports a number of predefined commands that all have a unique decimal value assigned to them. This value is placed in the command register by the kernel module before invoking the hypercall interrupt.

6.3.3.2 Function Call Translation

An intuitive way to transfer information to an external component is to use an output function. As an example consider the `printk` function, which is an output function that is often used within the Linux kernel and allows us to write information to the kernel log. If we could make use of output functions such as `printk` to transfer information to the hypervisor, we would obtain an output mechanism that is based upon existing functionality and thus easy to use. However, the problem that arises if we try to use output functions within an injected module is that the output function will be executed within the VM. Thus the output of the function will actually end up within the guest instead of on the host. This will not only violate our stealth requirement (S3), but also means that modules which use output functions cannot be used for injection without major modifications, violating one of our functional requirements (F1).

To solve this problem our system makes use of a technique that we refer to as *function call translation*. The main idea of this technique is to translate function calls occurring within the guest such that they can be executed on the host instead. To enable function call translation for a specific output function contained within a module, all calls to the output function will – similar to all other external function calls – be replaced with calls to a wrapper function. From this point on whenever the output function would be called in the original module, the wrapper will be invoked. This wrapper function will make use of the general communication channel provided by our system to invoke the hypervisor component. However, before doing so the wrapper function will put the decimal encoding of the original output function into the command register. Based on this value the hypervisor component can therefore determine, which output function the injected module tried to invoke. Given that there is a function on the host that has the same function signature as the original output function within the guest, we can then produce the output on the host by calling this function with the exact same arguments as the original output function. To achieve this, we first have to extract all function arguments that were provided to the original output function from within the guest. Depending on the architecture and the guest OS, these arguments will either reside on the guest's stack and/or within general purpose registers of the guest.

Once we have obtained the arguments from the hypervisor, we need to prepare them for the use on the host system. While numeric data types can simply be forwarded to the output function on the host, this is not the case for pointer types. This is due to the fact that the pointers will contain guest virtual address that are valid for the guest system, but not for the host system. Thus we have to translate the virtual address contained within the pointer arguments to the corresponding virtual address on the host system. Notice that this approach is possible, since the physical memory area that is used by the guest is actually managed by the hypervisor and corresponds to a virtual memory area on the host system. Therefore every virtual address within the guest is also accessible through a host virtual address.

When the translation of the arguments has been completed, we have to place the arguments in the correct stack and/or register locations on the host such that they will be used for the next function call. Finally, we can invoke the output function on the host. Since the output function receives the exact same arguments as the original output function, it will generate the same output given that it provides the same functionality.

X-TIER currently uses function call translation to translate all calls to output functions occurring within an injected module to calls to output functions that are executed on the host system instead. In particular calls to `printk` (Linux) and `DbgPrint` (Windows) functions that are executed by an X-Module are translated to calls to `printf` on the host system. Notice, however, that the proposed mechanism of function call translation is general and could be applied to arbitrary function calls. Moreover function call translation is completely transparent to the developer of a kernel module (F1).

6.4 Evaluation

We implemented a prototype of our framework for the x86 architecture that is based on the Linux KVM hypervisor. In this section, we make use of this prototype to evaluate our framework against the functional and security related requirements that we defined in Section 6.2. In this process, we will also discuss the functional restrictions of our current prototype and the security related issues of invoking external functions. In addition, we will demonstrate the capabilities of X-TIER at the hand of several example applications.

Name	*Description*
`tasklist`	Shows the running processes.
`lsmod`	Prints a list of the loaded modules.
`netstat`	Displays the open TCP and UDP connections for each process.
`files`	Prints a list of all open files for each process.

Table 6.1: The kernel modules that were used to conduct the performance evaluation.

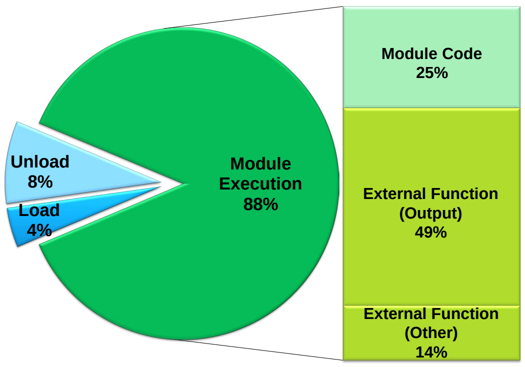


Figure 6.3: The average execution time distribution of all modules shown in Table 6.1.

6.4.1 Performance

We used four different kernel modules that extract typical security relevant information from within a guest system to empirically evaluate the performance of X-TIER. The name of these modules as well as a description of their functionality is shown in Table 6.1. Each of the modules was implemented for Linux and Windows. To test function call translation, each module was designed to print the information that it obtains using `printk` (Linux) and `DbgPrint` (Windows), respectively.

For the purpose of implementation, compilation, and injection of the kernel modules, we used two VMs. The virtual hardware configuration of both VMs consisted of a single virtual CPU, 512 MB of guest physical memory, and a 20 GB virtual hard disc. As OSs we chose the 64-bit version of Ubuntu 11.04 Server and the 32-bit version of Windows 7 Professional SP1. We purposely selected a 64-bit OS and a 32-bit OS for the VMs to verify that our framework is generic enough to handle both system types. The host OS was Ubuntu 12.04 64-bit running on a machine with an Intel Core i7-2600 3.4 GHz CPU and 8 GB RAM.

The modules were compiled with gcc 4.6.3 (Linux) and the Build Utility 6.1 (Windows), respectively. While we used the default compiler flags to compile the Windows modules, we compiled the Linux modules with the option `mcmodel=large`. This flag instructs the compiler to reserve 8 bytes for each address within the module code instead of 4 bytes. Although this option is not required, it allows us to inject a Linux kernel module anywhere within the 64-bit virtual address space.

To measure the performance of X-TIER, we compiled (Linux) or respectively extracted (Windows) the Linux 3.6 Kernel Image, a 467 MB tar file, within the guest. In the process, we repeatedly injected one of the modules into the VM at intervals of one second. The information that was obtained by an injected module was printed on the host system using function call translation. In the process, each call to an output function within a module lead to an individual VM exit. No output data was buffered within the module to increase the performance. This experiment was repeated for each of the modules and the resulting runtime overhead was measured from the hypervisor.

The results of the experiments are shown in Table 6.2. On average, each module was injected 2,950 times on Linux and 1,078 times on Windows. The highest performance overhead was introduced by the `files` modules, which incurred an overhead of 1.08% and 2.76% respectively. The reason for this is that these modules had the longest runtime within the VMs. The more time a module requires to execute, the longer all other code within the guest will be frozen and consequently the higher will be the resulting performance impact. As Figure 6.3 shows, the runtime itself is heavily influenced by the number of external functions that a module invokes. This is a result of the fact that the invocation of an external function leads to at least one VM exit, which is a costly operation. Since output functions account for almost 50% of the runtime in the current implementation, we expect that the performance of our prototype could be considerably increased by buffering output data within the guest system instead of processing each call to an output function individually.

In summary, the experiments show that our system is capable of effectively bridging the semantic gap by injecting normally created kernel modules from the hypervisor (F1). The overhead of the approach is very small even if a module is frequently injected into a VM. The performance impact of the injection mainly depends on the execution time of the injected module, which in turn is influenced by the number of external functions that the module invokes.

6.4.2 Security

To achieve our goal of a secure execution environment (G1), we make use of VMI and strongly isolate security applications from the machine they try to protect. This isolation ensures that a security application on the hypervisor level cannot be accessed by code running in a VM. However, by injecting a module into a VM and executing it within the context of the guest, we break the native isolation that VMI provides. Ideally, a module that was injected into a VM should have the *exact same* security properties that it would have if it was running outside of the guest. X-TIER achieves this by making use of *runtime isolation* and *function call unmapping*.

Runtime isolation restricts the execution of the guest system to the injected module and the guest code that it invokes. Any code that is not explicitly required by an X-Module will be frozen during its execution. This effectively isolates the module within the VM. Even in the case that the X-Module is faulty, exceptions will not be handled by the guest system, but on the hypervisor level as they would if the module would run outside of the guest. The only way to disable the proposed lightweight isolation mechanism is to reenable the timer-interrupts within the guest system by setting the IF flag, which is the *only* mechanism that is used by our system that cannot be protected from the hypervisor.

Experiment		*Runtime* [ms]			*Result*			
OS	*Module*	LOAD	EXEC	UNLOAD	IN	FUNC	OUT	*Overhead*
Win	`tasklist`	0.11	1.21	0.16	1,047	0	31,531	**0.15%**
Win	`lsmod`	0.11	7.21	0.16	1,051	140,975	144,128	**0.80%**
Win	`netstat`	0.10	2.93	0.23	1,182	0	30,322	**0.28%**
Win	`files`	0.13	25.38	0.16	1,030	469,939	528,131	**2.76%**
Linux	`tasklist`	0.30	2.43	0.59	2,925	0	209,581	**0.18%**
Linux	`lsmod`	0.30	0.49	0.61	2,957	0	38,990	**0.05%**
Linux	`netstat`	0.30	0.45	0.64	2,967	0	21,603	**0.05%**
Linux	`files`	0.32	12.37	0.79	2,954	523,098	451,233	**1.08%**
Linux	`LxS`	0.24	5.18	0.52	28,330	3,944,284	89,058	**4.30%**

Table 6.2: Results of the experiments. The columns show for each module the average runtime of the injection (LOAD), execution (EXEC), and removal (UNLOAD) phase in ms, the total number of injections (IN), of external function calls (FUNC) and calls to output functions (OUT), and the total overhead that the modules' injection incurred.

However, due to the fact that the injected module controls the virtual CPU, this can only be done by the module itself. Therefore we do not consider this to be an issue.

While runtime isolation is sufficient to protect the normal execution of an injected module, it cannot ensure a module's isolation if the module itself invokes an external function. To solve this problem, X-TIER temporarily unmaps an X-Module whenever it invokes an external function. This is realized with the help of the EPT. In particular, X-TIER removes the guest physical memory pages that an X-Module occupies from the EPT. As a result, the guest will no longer be able to access the module. Since the EPT can only be modified by the hypervisor, the proposed mechanism of function call unmapping can reliably isolate a module during external function calls. To be useful in practice, however, our system does not extend this isolation to the function arguments. That is, an external function will be able to access and modify its function arguments. While the access of an external function will be restricted to the function arguments alone, this provides a small attack surface. This attack surface, however, is not limited to our system, but rather inherent to the problem of invoking untrusted code. In fact, if the module would reside on the hypervisor level and would invoke a function within a guest system, the same problem would exist. Nevertheless, our system reduces the attack surface by using individual wrapper functions for each external function. Rather than updating all arguments after an external function call, each wrapper only updates the function arguments within a module's data region that a specific function is supposed to modify. Consequently, an external function can only access and update the memory regions that it requires to fulfill its purpose.

Finally, it is worth to emphasize that the execution of an injected module leaves no traces within a guest system unless the module purposely modifies the state of the guest (S3). This is due to the fact that our system only operates on memory regions that will be removed once a module finished its execution. As a result, an injected module that constrains itself to only reading data structures within a guest system can only be detected based on timing attacks, since it is atomically executed and leaves no traces. Note that, as described in Section 2.3.2.1, timing attacks are an inherent problem of *all* VMI-based approaches and thus represent a limitation that is independent of our framework.

6.4.3 Example Applications

To highlight the capabilities of X-TIER, we created multiple example applications that demonstrate the qualities of our system. Since the main motivation behind X-TIER was to provide a secure and flexible foundation for malware detection and removal, we created a virus scanner for Linux as well as multiple malware removal modules to illustrate how well our framework fulfills this role in practice. In addition, we implemented an external hypervisor-based shell called X-Shell that enables its user to execute Linux shell commands on the hypervisor level which are then automatically redirected into a VM. This provides further evidence for the fact that X-TIER is indeed a powerful framework with a broad scope of applications. In the following, we will cover each of these applications in turn starting with the virus scanner that we implemented.

6.4.3.1 Virus Scanner

To demonstrate the possibilities of X-TIER in connection with event interception (G3), we implemented an on-access virus scanner for Linux using X-TIER that we call LxS. This virus scanner consists of two parts: a hypervisor component and a kernel module. The kernel module is injected by X-TIER every time a file is executed within the VM using the `execve` system call. In this particular case, we trap this event by setting a debug breakpoint on the address of the `execve` system call using the debug registers of the x86 architecture. Notice, however, that this mechanism is functionally independent of X-TIER. X-TIER can be combined with arbitrary software or hardware-based trapping mechanisms. Researchers presented many such mechanisms for VMI over the last years. A good overview over some of these techniques for the x86 architecture can be found in [119].

Once the kernel module has been injected, it reads the file that should be executed, calculates its SHA-512 hash, and transfers the file name as well as the SHA-512 hash to the hypervisor component using function call translation. The hypervisor component will then compare the calculated hash to a virus database and signal to the injected module whether the file is malicious or benign. In the first case, the module will deny access to the file by returning an error code, while the module will invoke the original `sys_execve` function in the latter case, which will trigger the execution of the file. This mechanism is completely transparent to the guest OS and cannot be evaded.

We tested the above described virus scanner once more by monitoring the compilation of the Linux kernel. Thereby we used a clam-av database that contained 45,039 SHA-512 hashes to check the executed binaries. The results of this experiment are shown in Table 6.2. As one can see, the virus scanner application tested 28,330 executables during compilation and only incurred an overhead of 4.30%. In addition, we verified the detection mechanism of the approach by executing several malicious files including the adore-ng rootkit, the suckit rootkit, the mood-nt rootkit, and the enyelkm rootkit. We choose these rootkits as they are often used for testing purposes in the research community (e.g. [48, 128, 183]). In all cases, the access to the malicious files was denied. No false positives were observed.

6.4.3.2 Malware Removal

An important issue that we have not discussed so far is malware removal. Instead of solely detecting malware infections, we want to go beyond such approaches and also provide the possibility to react to infections, which is a crucial functionality that is often overlooked. While detecting malware infections is the first step, we must also remove the infection to restore a clean system state. Consequently, if our framework would only provide a basis for malware detection, but not for malware removal, it could never be effective against malware in the real world, since it would leave the job halfway done.

To test whether X-TIER is suitable as a basis for malware removal, we created individual defense modules for the adore-ng rootkit, the suckit rootkit, the mood-nt rootkit, and the enyelkm rootkit. Each module was thereby designed to detect its corresponding rootkit in memory using a signature[3]. Once detected, the modules would remove the rootkits entirely from the infected system by reverting all the changes that the rootkits conducted during infection. The necessary information was thereby obtained through manual analysis of the rootkits, as is usually the case in the real world. Since modules injected by X-TIER can make use of all exported kernel functions and data structures, the creation of these modules turned out to be no more complex than writing a basic kernel module. In addition, because all the created modules where not designed for the specific use with X-TIER, but rather as normal kernel modules, each of the created modules can be injected with X-TIER or be run directly from within a guest system. This provides additional evidence for the fact that X-TIER is indeed able to inject arbitrary kernel modules from the hypervisor.

For testing purposes, we sequentially infected the Ubuntu 11.04 Server VM with each of the rootkits. We then executed each of the defense modules using X-TIER. Due to

[3]Note that we purposely chose a signature-based approach, since signature-based detection is still the most common approach to malware detection. This demonstrates that X-TIER could be used as a basis for current detection and removal mechanisms. However, naturally, our framework could also be employed to realize other detection mechanisms as we will show in Section 7.4.

the signature-based approach, the defense modules only triggered when the machine was indeed infected with the corresponding rootkit. In addition, the defense modules successfully reverted all of the changes conducted by the rootkits. To ensure that the system remained stable after the removal of a rootkit, we left it running for 60 minutes once removal was complete. We did not observe any system instabilities during our experiments. The average runtime for a single execution of a defense module was 67,21 milliseconds. The main part of the execution time was thereby spend searching through memory for the rootkits with the help of the signature.

Finally, it is worth mentioning that we did not take any precautions for the case that a rootkit was currently executing when the defense module was run. In this case, it could occur that the system crashes once the rootkit is removed, since the defense modules also remove the code of the rootkits from memory. However, this situation did not occur during our experiments. To avoid this scenario, one could verify whether the IP pointed to the rootkit before removing it from memory.

6.4.3.3 X-Shell

Overview. To demonstrate the wide range of possibilities that our framework provides, we implemented a *hypervisor shell* based on X-TIER similar to EXTERIOR [48]. This shell essentially allows its user to execute almost arbitrary shell commands within the guest system from the hypervisor level. To accomplish this we make use of *system call redirection*. The key idea behind this approach is to redirect the systems calls executed by a process running on the host system[4] into a guest system. As a result, the process will behave as if running within the guest even though it is executed externally. In particular, it will produce the exact same information that it would produce if it was executed within the guest on the hypervisor level. To illustrate this approach let us take a look at a concrete example: `ls`.

The purpose of the Linux program `ls` is to *list* all files contained in a given directory. To provide this functionality `ls` relies on the `getdents` system call, which obtains the contents of a directory on Linux. Let us assume for the moment that the host uses the exact same OS as the guest. Consequently, if we execute `ls` on the host system, intercept the execution of the `getdents` system call, and inject the system call into the guest, `ls` will list the directory contents of the guest system instead of the host. The reason for this is that `ls` does not obtain the necessary information itself, but rather uses an interface for this purpose: the ABI. Since the guest and the host implement the same interface in our example, it becomes possible to redirect system calls from one system to another. This is essentially comparable to exchanging the component that implements the interface, which is possible as long as the interface remains the same.

[4]Note that the same approach could also be used to redirect system calls from one guest system into another. However, to ease the understanding, we here only consider the redirection from a host system into a guest system.

While powerful, this approach seems to have one major limitation: the target system must provide the same ABI as the system the application is executed on. This does not have to be the case though. By introducing a *virtualization layer*, we can provide the virtual ABI interface that a process expects and can at the same time support arbitrary ABI implementations. To achieve the latter, the virtualization layer must translate ABI invocations from the expected interface to the target interface and vice versa.

Implementation. To actually implement system call redirection, we have to solve four subproblems:

System Call Interception First of all, we obviously have to find a why to intercept the system calls of an application at runtime to be able to perform redirection.

System Call Selection Second, we have to decide whether a specific system call must be redirected into the guest or not. While one might think that every system call must be redirected, this is not the case in practice. For instance, when an application prints output this information should be printed on the host system and not on the guest system. Otherwise we would never see the output of the `ls` command, for example.

System Call Redirection Third, we must find a way to perform the actual redirection of the intercepted system calls into the guest system.

System Call Translation Fourth and finally, to provide a universal approach, it must be ensured that the host system can differ from the guest system. For instance, we may want to use `ls` to obtain information from a Windows guest. This requires a virtualization layer that translates system calls between different OSs.

Intercepting system calls essentially means that we must redirect the execution flow of an application whenever a system call is executed. One of the possibilities to achieve this is to add instrumentation code to an application during runtime. A tool that can be used for this purpose is PIN[5]. PIN relies on binary instrumentation to rewrite the code executed by an application at runtime. This approach allows us to add instrumentation code to each system call without actually modifying the application whose system calls we want to intercept. For this purpose, we created a PIN tool that invokes X-TIER whenever the monitored application executes a system call. The monitored application can thereby be an arbitrary ELF binary such as `ls`.

[5]https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

	Runtime [s]		*Results*		
PROGRAM	NORMAL	REDIRECTED	SYSCALLS	INJECTIONS	OVERHEAD
find	0.081	0.553	218	17	**6.80**
cat	0.039	0.309	42	6	**7.93**
netstat	0.007	0.172	54	5	**25.00**
ls	0.003	0.250	78	7	**83.33**
grep	0.004	0.454	96	8	**118.97**
uptime	0.005	0.880	95	21	**195.56**

Table 6.3: Performance results of our X-Shell, a hypervisor shell implemented on top of X-TIER. For each application the columns show the average runtime of the program with (REDIRECTED) and without (NORMAL) system call redirection, the number of system calls the program executes (SYSCALLS), the number of injected syscalls (INJECTIONS), and the total overhead.

Once X-TIER is invoked, our X-Shell component has to determine whether the current system call should be redirected or not. While this seems like a complicated problem at first glance, it turns out that a few manual rules created by an expert are sufficient to perform this step. In particular, we currently make use of twelve handwritten rules for system call selection. Based on these rules we are able to successfully redirect 70% (78 of 112) of the standard applications such as `ls` contained within the `/bin/` directory of our test system. Note that the selection rules are only dependent on the host system, but not on the guest system. Thus, a single rule set is sufficient to support different guest OSs. This is due to the fact that system call selection occurs before the redirection process.

To perform the actual redirection of the system calls we leverage X-TIER. In particular, we created a kernel module for each system call. Once injected, a module will execute its system call within the guest system and transfer the obtained information to the hypervisor. System call redirection can then be achieved by injecting the kernel module that corresponds to the system call that one wants to execute.

Finally, we must translate system calls between different OSs. For this purpose, we created a translation component that resides between PIN and X-TIER. Before a system call is injected, this components translates the system call and its arguments to a given target system. Each system call is thereby translated individually. Currently, we support only Linux guests. However, tools such as wine[6] have shown that the approach is general enough to be applicable to Windows as well.

Experiments. We tested the performance of our X-Shell by redirecting the execution of various common Linux tools on our host system into the guest and recording the overhead. Each application was thereby executed a hundred times. All experiments were

[6]http://www.winehq.org/

executed on a machine with an Intel Core i5-2520M CPU and 8 GB memory. The host system was a Gentoo Linux running kernel version 3.16, while the VM ran a Debian Sid Linux with kernel version 3.14. The average performance results of the experiments are shown in Table 6.3.

As one can see, the overhead of the approach is in general quite high ranging from a seven times slowdown up to a 196 times slowdown. What is interesting though is that the overhead not necessarily depends on the number of redirected system calls. In the case of `find` and `uptime`, for example, we redirected almost the same number of system calls (17 and 21 respectively). The overhead varies significantly though. Consequently, the performance of the approach also heavily depends on the type of system call that is redirected. In particular, the more information a system call produces, the higher will be the resulting overhead, since each data transfer leads to an individual VM exit. Finally, we like to stress that while the overhead seems high at first glance it is actually passable in practice. `uptime`, which is the program with the highest overhead, still executes in less than a second, for example. This is sufficient for most practical scenarios.

Summary. With the help of X-TIER, we are able to implement a hypervisor-based shell capable of executing programs from the hypervisor within a guest system. Our approach leverages system call redirection, a powerful technique that allows us to achieve this functionality without modifying existing applications. By combining system call redirection with system call translation, we could even support multiple OSs using X-TIER's OS independence as a foundation. Consequently, the resulting shell provides an intuitive, elegant, and universal way to control VMs from the hypervisor. This underlines the capabilities and usefulness of our framework X-TIER.

6.4.4 Limitations

There currently exist two limitations within our prototype. First of all, our prototype is not yet able to operate on multi-core systems. To support multi-core systems, the concept of runtime isolation must be expanded such that an X-Module cannot be accessed during its execution from one of the other cores. A possible solution to this problem would be to disable the interrupts on all CPUs and to put the additional cores into busy waiting loops while the X-Module executes on a single core. As a result, all other CPUs would be idle during the execution of an X-Module, which will have a negative impact on the performance of the approach. However, as modules have a runtime that typically only consists of a few milliseconds, this performance reduction should not be significant.

Second, there is a single functional limitation that our system places on the creation of kernel modules. Since X-TIER injects a kernel module from the hypervisor without involving the guest OS, specific OS data structures that are related to the module itself will not be created within the guest OS. For instance, an X-Module will not be able to use the `__this_module` (Linux) or respectively the `DriverObject` (Windows) variable, because the internal OS data structures that usually exist for each loaded module will not be available for an X-Module. In fact, if these data structures would exist, the guest OS would be aware of the existence of the injected module, which is why they are not created by X-TIER.

6.5 Related Work

Lares [112] was the first VMI-based system that made use of an in-guest component. The primary goal of this component, however, was not to bridge the semantic gap, but to provide the possibility of actively monitoring the guest system. In particular, Lares enables security applications residing on the hypervisor level to place tamper resistant hooks into a guest system. When triggered, a hook transfers the control to the in-guest component, which in turn transfers control to the hypervisor where it is redirected to the security application that placed the hook. The in-guest component thereby essentially functions as bridge between the hypervisor and the guest system, while the security applications remain on the hypervisor level. X-TIER goes far beyond this work by providing the possibility to execute arbitrary modules securely within the guest.

The first approach that – similar to X-TIER – aimed to achieve the secure execution of an entire application within an untrusted guest system was Secure In-VM Monitoring (SIM) [141]. The key idea behind SIM is to create an additional address space for the security application within the guest system from the hypervisor. To isolate the security application, the hypervisor carefully creates the layout of this new address space. First of all, it ensures that the virtual memory area that the security application occupies cannot be used by the guest system. This is accomplished by marking the corresponding virtual memory areas as used within the address space of the guest. Next, it maps all of the guest's memory areas into the newly created address space. However, all of the guest's memory regions are mapped as non-executable. This effectively allows the security application to access all of the guest's code and data regions, while the guest is unable to jump to its own code within the address space of the security application as it is marked as non-executable. Finally, SIM intercepts all changes to the page tables from the hypervisor to ensure that the guest cannot manipulate this carefully created mapping.

To be able to switch between the address space of the guest and the security application, SIM makes use of so-called entry and exit gates. In contrast to all other code regions, entry and exit gates are mapped as executable in *both* of the address spaces. Therefore the gates can be used by the guest as well as the security application. As the name suggests, an entry gate will switch to the address space of the security application, while an exit gate will switch from the address space of the security application to the address space of the guest. To intercept events within the guest system, the security application places hooks into the guest's code regions. On invocation a hook will transfer control to an entry gate, which will lead to an address space switch and the execution of the security application. Since the switch of the address space does not require a VM exit, SIM can achieve good performance results.

In spite of the fact that SIM tightly restricts the access to the address space of the security application, its architecture still exposes an attack surface. Since entry gates must be able to switch the address space from the guest's address space to the address space of the security application, this switch can also be performed from any other code region within the guest. If an attacker manages to disable the paging protections before performing the switch to the secure address space[7], it will become possible to execute arbitrary code in the secure region. This problem does not exist for X-TIER, since the injected code is *never* executed in parallel, but isolated within the guest. Whenever guest code is run, the module is removed from the system. This leads to a much simpler and less error-prone design, since the security of X-TIER does not depend on the paging protections of the guest system.

In addition, while SIM and Lares circumvent the semantic gap, they do not support the extraction of information from the guest system. That is, both approaches place an in-guest component into the VM, but the in-guest component is actually only able to access data structures or functions, if the component itself knows where they are located and how they look like. Neither Lares nor SIM takes care of external symbol resolution. As a result, the in-guest component must inspect the guest OS in the same manner as an out-of-guest approach. This eliminates the biggest advantage of the in-band approach: avoiding the semantic gap by making use of the guest's own functions to extract information. Lastly, both approaches are static, meaning that the in-guest component is not loaded dynamically into the guest as in the case of X-TIER. Instead, both approaches rely on a "trusted" kernel module for initialization. Consequently, the approaches are unsuited for scenarios where it is unknown whether the guest system has been compromised or not. This situation, however, frequently occurs in malware removal scenarios.

In contrast to SIM and Lares, process implanting [54] is capable of extracting information of a guest system and transferring it to the hypervisor. This is achieved by injecting a process from the hypervisor into the guest system. To provide stealth, the hypervisor does not create a new process within the guest, but rather substitutes the image of an existing process with the image of the program that is injected, a technique also known as *process hollowing* [177]. As a consequence, whenever the victim process is scheduled, the guest system will actually execute the injected program instead of the original process. The injected process can then access guest information using system calls and transfer the obtained information to the hypervisor with the help of hypercalls.

The security of the approach is based on the assumption that the guest OS system is trusted. To protect the implanted process against other malicious processes, its rights are elevated to root and it is protected from kill commands. In addition, the hypervisor

[7]Notice that this is generally possible for a guest OS. During boot, the guest OS must, for instance, be able to switch between different paging modes. If the hypervisor does not restrict this feature later on, the guest can switch between the paging modes as it wishes.

creates a new physical memory region for the injected process at runtime. Since other processes are unaware of this memory region and due to the fact that the implanted process is injected into a randomly selected guest process, malicious processes on the system neither know, which physical memory range the implanted process uses nor which process was substituted. While this does not directly protect the injected process, it makes it more difficult for an attacker to detect the implanted process.

The most significant drawback of process implanting is the restriction that the approach is only secure if the guest OS is with integrity. One of the main reasons to make use of virtualization, however, is to be able to protect security applications in spite of the fact that the OS kernel is compromised. In fact, if the OS is trusted, there is – from a security standpoint – no reason to resort to virtualization in the first place. Instead, we could simply make use of a security process within the system that is protected by the OS. Consequently, the technique is, in our opinion, not well-suited for the implementation of a framework for malware detection and removal. In contrast, to process implanting, X-TIER was designed to be able to execute modules securely within a guest even if the OS has been compromised.

While X-TIER provides strong security guarantees, one might argue that a compromised OS still can provide false information to an injected module, if the module performs an external function call. SYRINGE [18] tries to solve this problem by verifying the integrity of in-guest functions before and during their execution. In particular, SYRINGE enables hypervisor-based security application to inject function calls into a guest system. Before an in-guest function is invoked from the hypervisor, SYRINGE verifies the integrity of the function by comparing the hash of the code page containing the function's entry point against a whitelist. During execution, SYRINGE will repeat this process whenever the function leaves the current page and starts executing from a different page. In addition, the system ensures runtime control-flow integrity by verifying the targets of all `call`, `ret`, and indirect branch instructions according to a control-flow policy. To provide protection against runtime modifications of stack and heap data, SYRINGE executes the injected function call also atomically within the guest.

Since SYRINGE only injects function calls into a guest system, security applications are isolated by the hypervisor and must not directly be protected by SYRINGE. Similar to X-TIER, injected function calls are executed atomically within the guest system. In addition, control-flow integrity mechanisms are leveraged to ensure the integrity of the executed code. While this certainly raises the security of the approach, recent research has shown that control-flow integrity mechanisms can often be bypassed as we have described in Section 4.4.3. Further, the approach can at best ensure the integrity of the executed code, it cannot ensure the integrity of the data that is used by the injected function call. Therefore an attacker can still provide false information to the monitoring application, by performing DKOM attacks [152], for example. Additionally, checking the control-flow integrity of every invoked function leads to a significant performance overhead. In our opinion, the increase in security that control-flow integrity provides, does not justify this overhead. On the contrary, control-flow integrity may suggest a false sense of security. In general, one should consider all information obtained from an external source as *unreliable*. Nevertheless, enables X-TIER injected modules by its design to implement arbitrary security checks such as control-flow integrity verification if desired. Effective mechanisms can even be shared among modules by simply including the source code.

Finally, SYRINGE can in contrast to X-TIER not provide security applications with access to the kernel's data structures. It only provides access to the kernel's functions. Data structures, however, represent a second important source of information.

6.6 Summary

In this chapter, we presented X-TIER, a framework that allows security applications to inject kernel modules into VMs at runtime. By employing runtime isolation and function call unmapping our system is able to execute an injected module securely within the context of an untrusted guest system. In the process, injected modules have access to all exported guest OS data structures and can even invoke guest OS functions without sacrificing isolation or compromising their security.

Prior to injection, X-TIER converts modules into our uniform X-Format. This step requires no changes or recompilation of existing kernel modules and allows our system to support multiple OSs while remaining extensible. In addition, X-TIER provides an intuitive communication channel that allows injected modules to send and receive information to and from a security application residing on the hypervisor level.

Our prototype implementation of X-TIER is capable of injecting kernel modules into Windows and Linux guests. The evaluation of our system shows that the performance impact of module injection is small even for frequently injected modules. Due to its functionality, security, and performance, X-TIER is very well-suited for creating hypervisor-based security applications as we demonstrated with various example applications.

In conclusion, our framework enables hypervisor-based applications to securely and completely bridge the semantic gap, which is the key problem that all VMI-based application face. By providing the possibility to inject unmodified kernel modules, X-TIER is in addition easy to use, extremely flexible, and widely applicable. Due to these capabilities, our framework is able to provide the secure and flexible foundation for malware detection and removal that we requested at the beginning of this chapter. Equipped with this framework, we can now counteract the threat of data-only malware.

Chapter 7

Countermeasures

Having provided a secure and flexible basis for malware detection and removal with our framework X-TIER, we will in this chapter discuss potential countermeasures against data-only malware. To provide a comprehensive approach, we will thereby consider defenses against this malware form on a conceptual as well as on a technical level. To this end, we will first analyze the defense model that is commonly used on systems today and explain its shortcomings. Based on our observations, we will introduce an alternative model for system defense that employs a defense in depth approach to counteract malware in different stages of its execution. The key idea behind this model is to install defense mechanisms on multiple layers within the system in order to create a network of defense mechanisms. Following this idea, we create a defense network against data-only malware by discussing countermeasures on each of the layers that the model proposes. In the process, we will particularly focus on the detection of data-only malware and present three specific technical countermeasures against it. To make evasion more difficult, each of the countermeasures is thereby based on X-TIER and exploits one of the inherent dependencies that we identified during our analysis of data-only malware in Chapter 3 for the detection of the malware form. By combining an elaborate defense strategy with strong technical countermeasures, we can provide effective initial defense mechanisms against data-only malware.

Chapter Outline. We begin by addressing the threat of data-only malware on a conceptual level. To this end, we introduce a system defense model in Section 7.1. Based on this model we state our defense strategy in Section 7.2. The key idea behind this strategy is to counteract the threat of data-only malware on the three different defense layers that our model proposes: prevention, detection, and containment. Moving from the conceptual to the technical level, we apply this approach and discuss possible prevention mechanisms against data-only malware in Section 7.3. In Section 7.4 we advance to the second layer and present three individual detection mechanisms for data-only malware. Each of the mechanisms is thereby based on one of the fundamental dependencies of the malware form. During our discussion, we illustrate the effectiveness of each mechanism with detailed experiments. From the detection layer we then move to the last layer of our defense model, the containment layer, in Section 7.5. Finally, we summarize the chapter in Section 7.6.

7.1 System Defense Model

In the previous chapter we introduced a secure framework for the detection and removal of (data-only) malware in the form of X-TIER. However, X-TIER "only" provides a secure *technical* basis for system defense. To counteract the threat of data-only malware, it is crucial to address the problem on a *conceptual* level as well. In particular, we must define a *system defense model* that outlines how our defense against data-only malware should actually be implemented. For this purpose we will first take a look at the predominant defense model found on most systems today and explain its shortcomings. Based on our observations, we will then discuss how we can mitigate these shortcomings using a different model for system defense.

As discussed in Section 2.3, the present system defense model on the end host is based in its entirety on the OS. All processes including security applications such as antivirus software rely on the OS for protection. The OS thus represents a *single point of failure*, which is the first fundamental flaw of the model. If an attacker should manage to compromise the OS, the security model collapses and all security guarantees are lost. As a consequence, it is crucial that the OS can never be infected by malware. Ensuring this is the responsibility of malware protection and detection mechanisms. And therein lies the second fundamental problem of the model: it is based on the assumption that present protection and detection mechanisms are capable of *preventing* malware infections entirely. As we saw in Chapter 4, current defense mechanisms can, however, not even remotely provide this capability. Moreover, considering that the malware detection problem is in general *undecidable* [29], it is doubtful that there will ever be a mechanism that can provide such guarantees. Consequently, our current defense model is doomed to failure. To solve this problem, we propose to make use of a different system defense model that does not solely rely on the OS for protection, but rather leverages multiple security layers. This model is shown in Figure 7.1.

As can be seen, the proposed model leverages the principle of *defense in depth* [165]. The main idea behind this approach is to increase the security of a system by using multiple levels of defense instead of just one. This approach forces an attacker to overcome *all* defense layers in an attempt to compromise the system, which is in general a much harder task than bypassing a single layer of defense. While this approach is beneficial for system defense in general, it is especially helpful for malware defense. This is due to the fact that many current countermeasures face practical limitations as we have observed in Chapter 4. By leveraging a defense in depth approach, we can effectively combine multiple defense approaches on multiple layers. As a result, a single malware defense

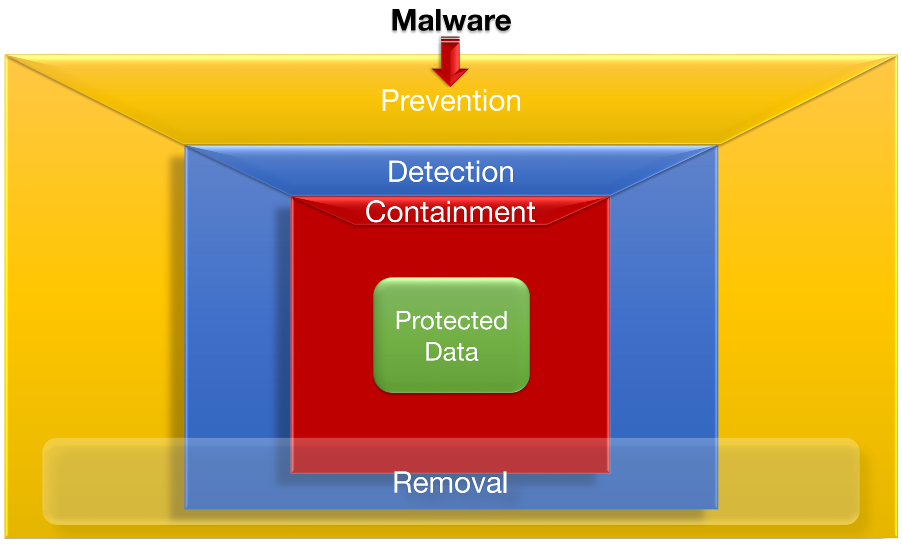


Figure 7.1: The system defense model that we leverage within the thesis. The model consists of four fundamental parts: prevention techniques (yellow), which try to prevent infections, detection techniques (blue), which are responsible for detecting infections, containment (red), which aims to limit the damage that an infection can incur, and removal (black), whose purpose is to remove infected files and to clean the system across all defense layers.

mechanism (e.g. antivirus) is no longer the sole security keeper on a system. Instead, it becomes a part of a *defense network*. If a single mechanism is evaded, there remain others that can take its place. That is, there is no longer a *single point of failure*. In addition, the approach allows us to better absorb drawbacks of individual mechanisms. For instance, anomaly-based detection often suffers from false positives. By combining this defense technique with other approaches, we can reduce the impact of false positives by only raising an alarm if multiple detection mechanisms are triggered at the same time.

In our model, we propose the use of three layers of defense that can in turn each consist of multiple individual defense mechanisms. Each layer thereby attempts to hinder (data-only) malware during a different stage of its execution. The first layer attempts to stop malware infections before they can actually occur, which would be ideal. This is why we refer to this layer as *prevention* layer. It is shown in yellow in Figure 7.1.

Since the prevention layer represents the first line of defense and aims to hinder malware before it can infect a system, it can similarly to existing security mechanisms rely on the OS for protection. Note, however, that we use the term “prevention mechanism” in this thesis in a broader sense than is usual. In particular, we assume that a mechanism must not necessarily *stop* the execution of malware a priori to qualify as a prevention mechanism. Instead, we distinguish between two types of prevention mechanisms: prevention mechanisms that detect malware *before* its execution and prevention mechanisms that thwart the *successful* execution of the malware. An example of the first category are signature-based detection mechanisms, while protection mechanisms such as ASLR belong to the second category.

The second defense layer of the model is formed by the *detection* layer shown in blue in Figure 7.1. Realistically it is impossible to prevent malware infections entirely. We therefore require a defense layer that detects malware infections that manage to get past the prevention layer. At this point, however, the reader my ponder the question whether this approach actually makes sense. Is it not already too late to detect malware once it has successfully been executed? After all in this case the malware already had the chance to fulfill its malicious purpose. What is important to understand though is that malware detection nonetheless remains essential. While we cannot undo any harm done, we can prevent *future* damage by detecting and removing the malware from the system. Consequently, malware detection remains an important defense layer independent of the fact whether the system is infected or not.

Since the task of the detection layer is to detect malware once it infected the system, detection mechanisms cannot rely on the OS for protection and must in fact be isolated from it such that they can remain functional even if the OS is compromised. For this purpose, we introduced X-TIER as a secure and flexible technical foundation for malware detection. Since X-TIER allows security applications to operate outside of the system they try to protect and can on top of that support the execution of existing security applications *without* modification, it provides a perfect basis for the implementation of such a detection layer.

The detection layer leads us directly to the last defense layer in our model: the *containment* layer (red). As stated above, it is highly unlikely that we can prevent malware infections entirely. An important measure for malware defense that complements our detection layer is therefore to implement mechanisms that reduce the damage that malware can conduct. The fundamental idea behind this approach is that if we cannot prevent malware infections, we can at least tightly limit their possibilities. In fact, if we could isolate the malware in the system such that it cannot affect any other application, it will in many cases matter little whether our system is infected with malware or not. As in the case of the detection layer, the protection of individual applications or their data can thereby be conducted from the hypervisor. We will discuss some of the potential approaches that can be leveraged for this purpose in Section 7.5.

While containment sounds like a strange idea at first glance, it is actually a quite logical step in the evolution of anti-malware mechanisms that we can currently also observe in the real world. For instance, Symantec, a very well-known anti-malware company, just recently declared that traditional antiviruses are "dead", because they are no longer able to adequately protect a system [132]. As a consequence, the company now focuses on minimizing the damage of malware infections. That is, they actually focus on malware containment instead of malware prevention.

The final element in our defense model that we have not covered so far is the *removal* layer (black). Naturally, we do not only want to detect malware, but we also want to take measures to remove the malware from the system once it has been identified. This step is crucial since it actually ensures that malware is no longer able to affect the system. The detection of the malware is only the first step in this direction. Just knowing that the system is infected, however, does not solve the problem. The removal of the malware does. Similar to detection and containment, the removal layer must thereby operate in isolation from the system it protects. As shown in Section 6.4.3.2, X-TIER also provide a flexible and secure basis for this layer.

7.2 Defense Strategy

Applying our system defense model, we will in the following create a network of defense mechanisms to mitigate the threat of data-only malware. For this purpose, we will discuss countermeasures on each individual layer that the model proposes. In the process, we will particularly focus on the detection of data-only malware, since it is the most important defense layer in the context of this thesis. To ensure that each detection component of our network is effective and difficult to evade, we will leverage the dependencies we identified in Section 3.6.3 as basis for our countermeasures. Since these dependencies cannot be easily resolved and are on top of that inherent to all types of data-only malware, they are predestined as a foundation for a comprehensive defense strategy.

To refresh reader, we identified the following dependencies during our analysis of data-only malware:

Application Dependency	Data-only malware always requires a host application to function.
Control Structure Dependency	Since data-only malware is a data-only program written in a code reuse language, it must always have a control structure that manages its execution.
Switching Sequence Dependency	Finally, before data-only malware can be executed, the virtual IP that it leverages must be set to the control structure. This is accomplished by executing a switching sequence.

With this knowledge in mind, lets take a look at our first defense layer: the prevention layer.

7.3 Prevention Layer

When we think about preventing the successful execution of data-only malware, a dependency that immediately comes to mind is *application dependency*. In particular, if we could prevent the malware from finding the gadgets it requires to execute, we would break the elementary concept that data-only malware is based on and stop it from functioning. A current defense approach that follows this idea is ASLR.

7.3.1 Rerandomization

ASLR randomizes the address space of an application to make it more difficult for an attacker to obtain the addresses of the gadgets she requires. One crucial advantage of ASLR is thereby that it only needs access to the binary representation of an application to function and can thus directly be leveraged to protect existing applications without modifying them first. As Snow et al. [148] have shown, however, the main problem of ASLR, even of fine-grained ASLR that operates on the instruction-level, are memory disclosure vulnerabilities that allow an attacker to leak addresses. An interesting approach to counter this attack would be to *rerandomize* applications during runtime [11, 148]. That is, to randomize the address space of an application not just *before*, but also during its execution. As a result, addresses leaked to an attacker will only provide a *temporary* view of the application. This temporary view, however, must not correspond to the actual state of the application. In fact, if the rerandomization is performed between the time an attacker obtains an address and an attack is launched, a leaked address will become useless to the attacker, since the layout of the application will already have changed when the attack is performed.

Due to the fact that addresses may constantly change, rerandomization represents a very strong defense mechanism against data-only malware and dynamic hooks. In fact, it targets one of the fundamental requirements that these attacks are based on: to create data-only malware and dynamic hooks we must be able to predict the addresses of instructions or objects within the system. While we require the addresses of instructions to actually implement the functionality of data-only malware, we must in the case of dynamic hooks predict the location of the value that we want to override beforehand. Rerandomization naturally complicates this task significantly. This is especially true for persistent data-only malware and dynamic hooks, since rerandomization may change the layout of an address space between invocations of the malware or the hook.

There are two basic approaches that can be used to perform rerandomization: one can either just rerandomize code regions or one can rerandomize both code and data regions. The former approach has the advantage that it can in general be easier implemented, since we only require the location of all control data used by the application to perform the rerandomization. However, on the down side, the method can only protect against data-only malware and dynamic control hooks. To counteract dynamic data hooks, we instead need to rerandomize data regions as well. For this purpose, however, we require the location of all pointers within the application, which leads to a much more complex design and a higher performance overhead as every rerandomization requires the modification of more values compared to just randomizing the code regions.

Rerandomization in Practice. To the best of our knowledge, there currently exists no binary-level implementation that randomizes code regions as well as data regions. However, Williams-King [179] only recently presented a practical implementation for rerandomizing the code regions of ELF binaries on a Linux system. To accomplish this, he makes use of of a two-fold process. In the first step, the code regions of the ELF executable are disassembled in memory. During this process the “Shuffler”, which is the component that performs the rerandomization, records all IP relative instructions and all branches to fixed addresses within the binary, since these addresses must be updated when the rerandomization is performed. In the second step, all `call` instructions within the executable are replaced with a `jmp` instruction to a trampoline function. The idea thereby is to avoid that return addresses are pushed on the stack, which would make rerandomization more difficult as the Shuffler would in this case need to unfold the stack in order to identify them. By leveraging a trampoline, the return addresses can be stored in a specific memory region where they can be updated during rerandomization.

While this covers most of the locations that must be modified in the process of rerandomization, there also exist various special cases that must be considered. For instance, may a binary contain function pointers or make use of C++ exceptions. Since the identification of such structures is difficult on the binary-level, Williams-King solves these cases by intercepting faults during the execution of the program and redirecting them to the appropriate location. Notice, however, that this represents a fundamental flaw in the current design, since this approach enables an attacker to invoke arbitrary functions and thus to perform ret2libc style attacks.

Williams-King’s implementation is currently based on ptrace, which enables the Shuffler to randomize the memory area of arbitrary processes. During rerandomization, the Shuffler updates the code region of the binary as well as all libraries that the binary uses. The runtime overhead of the approach thereby depends on the frequency of the rerandomization. When performing rerandomization every 50 ms, the average overhead measured was 11.3%.

To evaluate the security of the approach Williams-King studied the runtime of Snow et al.’s [148] just-in-time (JIT) code reuse attack under the assumption that an attacker requires at least 8 gadgets to perform a successful attack. Since Snow et al.’s results yielded that an attacker finds about 1,35 gadgets per page, Williams-King estimated that an attacker requires at least 71 ms for the attack in practice. Consequently, by performing rerandomization with a frequency of less than 71 ms, we can likely hinder most real world attacks.

Disadvantages. While rerandomization can be very effective against data-only malware and dynamic hooks, the approach of course also has some disadvantages. First of all, rerandomization naturally incurs a significant performance overhead (11% in the case of Williams-King’s approach), especially if not only code, but also data regions are rerandomized. Second, rerandomization cannot protect against brute force attacks. That is, if an attacker is able to repeat her attack arbitrarily, she will eventually guess the required addresses correctly and the attack will succeed. In fact, Shacham et al. [139] have shown that rerandomization only increases the security of a system against brute force attacks by a single bit compared to traditional ASLR. Thus it is essential that the search space for an attacker is chosen to be large enough that brute force attacks become impractical. Third and finally, special care must be taken that an attacker cannot predict the rerandomization. Consequently, a cryptographically secure random number generator should be used to determine the destination of the next randomization.

Summary. By rerandomizing applications at runtime, we effectively thwart an attacker from obtaining the addresses she requires to perform her attacks. Compared to ASLR, rerandomization can thereby even remain effective in the event of a memory disclosure vulnerability. However, rerandomization also decreases the performance of protected applications and performing rerandomization without loopholes is difficult in practice as can be seen based on the approach that Williams-King presented. While the approach solves many problems, it still enables an attacker to perform ret2libc attacks. In conclusion, rerandomization is no panacea, but, if implemented correctly, it could significantly improve the security of current systems against data-only malware and dynamic hooks.

7.3.2 Architectural Changes

Besides leveraging software approaches, we can also change the hardware layer to prevent data-only malware infections. In this section we discuss two architectural changes that we consider particularly effective against data-only malware: *encrypting instructions* and *isolating control data*.

Encrypting Instructions. Rerandomization makes it more difficult for data-only malware to obtain the addresses of the gadgets it requires to function. A hardware-based approach that goes into a similar direction would be to make use of an architecture that solely stores encrypted instructions within memory. During execution the CPU would then fetch the encrypted instruction from memory, decrypt it for its execution, and reencrypt it once it has been processed. As a consequence, the actual instructions that are executed would only be visible within the CPU.

When combined with ASLR, such a design would make it significantly more difficult for an attacker to find the gadgets for data-only attacks. Since the entire memory only contains encrypted data, even memory disclosure vulnerabilities would no longer be an issue for the security of the system. Given that the randomization is not predictable and the encryption is strong, the only option that remains for an attacker is to leverage blind attacks [11]. While such attacks are possible, they face even more restrictions than traditional data-only attacks and the attacks that have been presented so far ultimately also rely on memory disclosure to function. Consequently, the approach would be very effective against current data-only attacks, it would, however, also lead to a performance overhead due to the encryption and decryption of instructions.

Isolating Control Data. To infect a system, data-only malware must at some point in time modify the control flow of the host application. This is generally accomplished by overwriting control data such as return addresses. To make it more difficult to perform such attacks, control data could be stored in an isolated memory region that can only be accessed by specific instructions. For instance, a `call` instruction would only be allowed to store the return address at a free memory location within the special area, while a `ret` instruction could only obtain the last return address that was stored. As a result, non-control data can no longer be used to overwrite control data as both data types are stored separately from each other and access to the control data area is only allowed for a set of predefined instructions. In addition, it would become significantly more difficult for an attacker to load a control structure, since it consists of control data and must thus somehow be loaded into the specific memory area.

7.4 Detection Layer

The second layer of our defense model is the *detection* layer. The main purpose of this layer is to detect infections that bypass the prevention layer. Since this layer is particularly important for the defense against data-only malware, we will in this section present three general detection concepts for data-only malware. Each concept is thereby based on one of the inherent dependencies of data-only malware.

7.4.1 Detecting the Underlying Code Reuse Technique

Data-only malware relies on code reuse to function. This makes *code reuse techniques* to a key dependency of this malware form. Since code reuse techniques are on top of that generally only used by malware, it suggests itself to detect data-only malware based on the code reuse technique it leverages. That is, to detect data-only malware by looking for abnormalities in the system's execution that are caused by the characteristics of the underlying code reuse technique. For the sake of simplicity, we will refer to this approach as CRT-based (code reuse technique-based) detection.

Consider ROP, for example. ROP leverages the `ret` instruction to combine gadgets into a program allowing it to perform arbitrary computations. The original purpose of the `ret` instruction, however, is to *return* from a function call. Consequently, `ret` instructions actually do not appear randomly within a program's execution. Instead, their usage follows specific conventions. Most importantly, for each `ret` instruction there usually exists a corresponding `call` instruction. When the `ret` instruction is invoked, it will transfer control to the instruction following its corresponding `call` instruction. Thus if the corresponding `call` instruction for a `ret` is known, we can actually validate whether a `ret` instruction returns to its intended location or not. Since code generated by compilers in general follows the previously described conventions, a benign `ret` will in general return to its intended location. ROP, however, will violate these rules as the `ret` instruction is in this context used to provide the connection between multiple gadgets. As a result, there is no corresponding `call` instruction for the abused `ret`. This represents an anomaly that can be used to detect ROP and thus data-only malware.

While we have not delved deep into the approach of CRT-based detection yet, the example considered above already highlights a requirement of the approach: to be able to realize CRT-based detection, we require a mechanism that allows us to *monitor* the instructions that are executed by the target system. This is due to the fact that code reuse techniques often leverage low-level assembly instructions to function. In the case of ROP this instruction is the `ret` instruction, for example. To implement a detection mechanism we thus require access to these instructions. That is, we require an *instruction-level monitoring (ILM)* mechanism. Before we continue with our discussion of CRT-based detection, we will present such a mechanism. Based on this mechanism we will then propose a concrete approach to CRT-based detection of data-only malware.

7.4.1.1 Selective Instruction-Level Monitoring (SILM)

Requirements. While there are many different approaches that support ILM (e.g. emulation), the approach we are looking for must be suitable for malware detection. As such the approach must fulfill specific requirements:

R1 **Selective Monitoring.** ILM in general leads to a significant performance overhead. In order to keep this overhead as small as possible it is thus essential that we have the ability to *select* the instruction types we want to monitor. For instance, to detect ROP there is no need to monitor *every* executed instruction. Instead, it is sufficient to *solely* monitor `call` and `ret` instructions. Selective ILM provides this possibility.

R2 **Evasion-Resistance.** To be suitable for malware detection, it must be impossible for an attacker to *evade* the ILM mechanism. That is, if the machine executes an instruction selected for monitoring, this instruction must be received by the ILM mechanism.

R3 **Isolation.** Since the monitored system could according to our defense model already have been compromised when the detection mechanism is run, it is essential that the ILM mechanism is isolated such that it can remain functional even if the monitored machine is under complete control of the attacker.

R4 **Stealth.** To be useful for malware detection, the ILM mechanism must achieve a high level of stealth (i.e. it can only be detected based on side channel attacks as discussed in Section 2.3.3.1).

Taking these requirements into account, X-TIER, which we discussed in detail in Chapter 6, is perfectly suited as a basis for such an ILM mechanism as it natively fulfills the requirements R3 and R4. Thus we only need to extend X-TIER with the actual functionality required for selective instruction-level monitoring (SILM). For this purpose, we will in the following present a general approach for ILM from the hypervisor on the x86 architecture. In contrast to previous approaches, our technique is capable of *selectively* monitoring specific instruction types (R1), which enables it to achieve a much better performance than existing techniques. To accomplish this we make use of the performance monitoring counters (PMCs), a hardware feature that is nowadays available on almost all modern mainstream processors [88]. Since the PMCs can be protected from the hypervisor, our monitoring mechanism is not only flexible, but also stealthy (R4) and evasion-resistant (R2) [120]. Thus the approach provides a flexible and secure basis for the implementation of CRT-based detection approaches with good performance. Before we present our mechanism in more detail, we will, however, briefly discuss existing hypervisor-based ILM mechanisms and their weaknesses to highlight the need for a novel ILM approach.

Existing ILM Approaches. To the best of our knowledge, there currently exist three general hardware-based approaches that could be used to implement ILM from the hypervisor on the x86 architecture: page fault (PF) based ILM [119, 164], debug register (DR) based ILM [119], and trap flag (TF) based ILM [40, 102]. However, none of these techniques can provide the flexibility to monitor specific instruction types. To show this, each technique will be briefly described below.

Page-Fault (PF) based ILM. By making use of the page access bits of the EPT, it is possible to trap instructions that are either contained within (execute-disable bit) or try to read from (present bit) or write to (read-only bit) certain memory pages. Therefore by setting the desired access bits on every page, it is possible to trap all instructions or all instructions that involve memory operations on a guest system.

The main problem of using this approach for ILM lies in the fact that the mechanism is not working on an instruction basis, but rather on a page basis. Thus the mechanism is by design only able to trap all instructions contained within a certain virtual memory page or all instructions that have a memory operand accessing a certain virtual memory page. Consequently, PF-based ILM does not fulfill our requirement of selective instruction monitoring (R1).

Debug Register (DR) based ILM. The Intel x86 architecture provides four breakpoint DRs that can be used to trap instructions that are fetched from a specific memory address. By programming the DRs to contain the virtual memory address of every instruction that we want to monitor, we can implement an ILM mechanism.

In contrast to PF based ILM, this approach works on an instruction basis. However, the DRs are actually intended for setting hardware breakpoints. Therefore the only feature that the hardware supports in this case, is to raise an exception based on the locations specified in the DRs. The identification of these locations and the programming of the DRs has to be done in software. From this it follows that the complete logic that is required to implement a DR-based ILM mechanism, must actually be implemented in software. The role of the hardware is limited to the trapping itself, which the hardware cannot fulfill in all situations, due to the fact that hardware breakpoints that are placed directly after a `POP SS` or `MOV SS` instruction may not be triggered [64]. Thus DR-based ILM is not evasion-resistant (R2) [120].

Trap Flag (TF) based ILM. Using the TF for ILM is a very common approach which is, for example, adopted by Ether [40] and MAVMM [102]. The TF is a system flag, which will, if set, lead to the generation of an exception after every instruction that is executed by the processor. Therefore this hardware mechanism is the only one of the here presented mechanisms, that is actually intended for ILM. Besides, single-stepping the TF can in theory also be used to implement branch monitoring by combining it with the single-step on branches flag (BTF). However, this functionality cannot be used in practice, since the TF is modified by some guest OSs and the processor [64]. Once cleared, the processor will no longer generate exceptions and hence control will be lost. This also complicates single-step monitoring, but when every instruction is trapped, it is at least possible to reset the TF in case it was cleared between two instructions.

The whole problem arises due to the fact that the TF cannot be protected by the hardware. This means that there is no hardware mechanism which we are aware of that could be used to raise a signal if the TF is modified. Thus a TF-based ILM mechanisms is not only unreliable, but also not evasion-resistant (R2), since every process has access to its own TF and can therefore arbitrarily manipulate it.

Due to this problem Intel introduced the monitor trap flag (MTF), which is the equivalent of the TF on the hypervisor level. When the MTF is set, every instruction will cause a debug exception that is forwarded to the hypervisor. While this leads to an ILM approach that is evasion-resistant, the mechanism only allows to trap *every* instruction. Selective monitoring of specific instruction types is not possible (R1).

The Intel x86 Performance Monitoring Counters (PMCs). Most mainstream processors provide a performance monitoring unit (PMU) that enables applications to measure the performance of other applications or their own performance by monitoring the occurrence of specific hardware events. To reduce the overhead of the performance measurement and the lines of code that have to be added to an application that uses the PMU [153], an application usually does not monitor these events directly. Instead, the PMU provides so-called performance monitoring counters (PMCs) that count the occurrence of specific hardware events and can be accessed by the applications. To illustrate this mechanism, consider a PMC, for example, that counts the number of instructions that have been executed by the processor. By reading this counter multiple times and storing its value, an application can deduce how many instructions were executed in a certain period of time.

On the Intel x86 architecture, PMCs are MSRs that can in addition to the `RDMSR` and `WRMSR` instructions be accessed with the `RDPMC` instruction. Depending on the type of the PMC, the event that is counted by it is either fixed or programmable. While fixed PMCs count a specific hardware event that cannot be changed, programmable PMCs can be set to count one of the supported hardware events. For this purpose, there exists another MSR, the performance event select (PES) MSR, that determines the event that is counted by its corresponding PMC. Besides the event itself, the PES MSR of a PMC also provides additional controls that can be used to influence the counter. For example, it is possible to specify if an event is only counted at a certain processor privilege level and if the PMC should generate an interrupt when it overflows.

The number of fixed and programmable PMCs as well as the type of events that can be counted by a programmable PMC depend on the specific processor microarchitecture. On Sandy Bridge-based processors, for example, there are three fixed and four programmable PMCs [64]. When it comes to the countable events, it is necessary to distinguish between architectural events and non-architectural events. While architectural events are available on all x86 processor architectures, the number and type of non-architectural events depend on the specific microarchitecture. For example, on the Sandy Bridge microarchitecture, we find that there exist more than 200 non-architectural events in addition to the seven architectural events that are available across all Intel x86 microarchitectures [64].

PMC-Based Trapping. The idea behind PMC-based trapping is to trap the occurrence of hardware performance events that are counted with the help of programmable[1] PMCs to the hypervisor such that these events can be used to implement security mechanisms. As in the case of other security applications that try to monitor certain events within a VM from the vantage point of the hypervisor, the realization of such a mechanism requires two steps: First, it is necessary to force the hardware to generate a *signal* whenever the event occurs that we want to observe. Second, this signal must lead to a VM Exit that *transfers control* to the hypervisor.

In case of PMC-based trapping, the first step requires us to make sure that a PMC, which was set to count a specific hardware event, generates a signal when a certain number of events occurred. Since a PMC produces an interrupt on overflow, given that the corresponding flag was set in the PES MSR that controls the PMC, we can emit a signal if we force the PMC to overflow. This can be achieved by setting the initial value of the PMC to `MAX_PMC_VALUE -` $X + 1$, were X is the number of events that should occur before the overflow. For example, to cause an overflow after every counted event, we set the PMC to its maximum value. Thus the PMC will overflow when the next event is counted.

The interrupt signal that is emitted by a PMC on overflow depends on the setting of the local Advanced Programmable Interrupt Controller (APIC). Amongst other things, the

[1]A similar approach could also be realized using fixed PMCs, but we will ignore this case for the sake of simplicity.

Event Type	Description
ALL_BRANCHES	All branch instructions
CONDITIONAL	All conditional branch instructions
NEAR_CALL	All near call branch instructions
NEAR_RETURN	All near return branch instructions
FAR_BRANCH	All far branches

Table 7.1: Non-architectural events related to retired branches available on recent processors.

local APIC allows us to force the delivery of a non-maskable interrupt (NMI). Emitting a NMI has two advantages. First, the processor will handle the interrupt immediately, which will reduce the time that passes between the counter overflow and the moment the interrupt is handled by the processor. This is important due to the fact that it is possible that more than one event occurs during the time it takes to deliver the interrupt. We will defer the discussion of this issue to the paragraph after next where we consider PMC-based ILM in more detail.

Second, a NMI leads to a VM Exit if the appropriate flag is set within the pin-based VM-execution controls of the x86 architecture. Thus by using a NMI we can force a VM Exit and thus realize a control transfer to the hypervisor.

PMC-based Trapping for ILM. PMC-based trapping can be directly applied to ILM by making use of hardware performance events that are related to instruction execution. Since modern microarchitectures support events that allow to count specific instruction types, the resulting ILM approach is capable of selecting these instruction types for monitoring. To provide the reader with a better understanding of the type of events that are available, Table 7.1 shows a small subset of the non-architectural events related to taken branch instructions that are about to retire and are available on all modern x86 processors. In this context the term “retire” stands for instructions that have been executed by the CPU and whose changes will be committed to the architecture in the order in which they appear within the instruction sequence [64].

Counter Overflow Issues. Unfortunately, there is the possibility that more than one event occurs before the PMC overflow interrupt is received. The reason for this phenomenon is latency within the microarchitecture in combination with the speed of modern processors. Due to the latter it is likely that multiple instructions retire in a very short period of time, which also means that multiple monitored events could occur at nearly the same time. The PMC that counts these events will therefore be increased and will on an overflow generate a signal. This signal will then be forwarded to the local APIC that in turn will raise the selected interrupt. Because of latency, it is hence possible that more than one event occurs during the period of time between the PMC overflow and the delivery of the interrupt.

How many events can occur before the interrupt is delivered depends on the event that is monitored. The closer events appear to each other within the instruction stream, the higher are the chances that they will occur together before the interrupt is received. If we would implement a single-stepping mechanism with the help of a PMC by forcing an overflow after every instruction executed by a system, for example, it is very likely that more than one instruction would be executed before the interrupt is received. In fact, during our experiments with such a PMC-based single-stepping mechanism, we found that on the average about six instructions were executed before the interrupt was received.

Notice that similar issues could arise if PMC-based trapping is applied to other areas. How this problem can be solved must be considered on a case-by-case basis. In the following we will provide a solution for PMC-based ILM.

Instruction Reconstruction (IR). Because of the speed of modern processors and the latency within the microarchitecture, it is possible that we miss instructions that we want to monitor. However, the fact that we missed an instruction will not go unnoticed. On the contrary, the PMC that is used to count the instructions will tell us exactly how many instructions were missed, since it continues counting even after an overflow occurred. Therefore we know how many instructions we missed between the occurrence of the last interrupt and the current interrupt. This effectively reduces the general problem of recovering all missed instructions to the smaller problem of recovering all missed instructions that lie on the execution path from the last to the current interrupt. A possible solution to this problem is to reconstruct this execution path, reanalyze all executed instructions, and filter the instructions that we missed. To achieve this we will save the IP of the VM every time a monitoring related interrupt is received. This allows us to use the IP of the last interrupt as a starting point for the IR. To reconstruct the instructions, we will then sequentially decode all instructions that follow this starting point until we reach the current IP. A remaining problem are control transfer instructions that we encounter along the way, since the target of the control transfer may depend on memory operands that may have been overwritten in the meantime. However, since the control transfer instructions that we encounter were already processed during normal execution, we can make use of another hardware feature to recover the target locations of these instructions: the last branch record stack (LBR).

As the name suggests, the LBR is a stack that contains the last instructions that caused a control transfer. More precisely, the LBR consists of MSR pairs, where one MSR, the FROM MSR, contains the virtual address of the instruction that caused the control transfer and the other MSR, the TO MSR, contains the virtual address of the target of the control transfer. Therefore the LBR can be used to decide, if a branch was taken and what the resulting IP value was.

The size of the LBR depends on the processor that is used. Recent processor families usually provide a LBR that consists of 16 MSR pairs. The top of the LBR is indicated by a special MSR, the so-called top-of-stack pointer (TOS) MSR. Once the TOS reaches its maximum value, it will wrap around and the oldest entry on the stack will be overwritten. Therefore the number of branches that can be reconstructed with the help of the LBR, depend on its size. However, we do not expect this to be an issue in practice, since the size of the LBR was always sufficient for IR during our experiments. But even if we had to record more branches than the LBR can hold, this is not a problem, because it is also possible to use the branch trace store (BTS) instead of the LBR, which is basically a LBR in memory and can therefore hold as many branches as memory is reserved. Further in contrast to the LBR, the BTS can be programmed to generate a signal before it overflows, which means that branches can always be processed before they get overwritten [64].

By using the LBR it is finally possible to reconstruct the complete instruction stream by saving the TOS of the LBR in addition to the IP on every monitoring related interrupt. While moving through the execution path starting from the last IP we can then increase the last TOS whenever a control transfer instruction is encountered that is recorded on the LBR and continue sequential decoding from the destination address. This will eventually lead us to the current IP.

Evasion-Resistance. While the steps described above allow us to monitor instructions from the hypervisor, it is not yet clear if PMC-based trapping is evasion-resistant (R2) [120]. To achieve this property, it is necessary that none of the hardware registers that are used by a PMC-based trapping mechanism can be manipulated from within the guest. Since the processor-based VM-execution controls on the x86 architecture provide the possibility to cause a VM Exit in case a MSR is read or written or a `RDPMC` instructions is executed, read or write accesses to the PMCs can be intercepted by the hypervisor. In addition, because all control structures that are related to PMCs are MSRs as well, attempts to access or modify the PMC control structures can also be intercepted. Thus PMC-based trapping is evasion-resistant (R2) provided that the hardware does not contain any flaws and implements the above mentioned mechanisms correctly [64, 120].

Summary. By leveraging the PMCs for ILM we obtain an isolated (R3) SILM mechanism that enables us to monitor the execution of specific instruction types (R1) within a guest system from the hypervisor. Since any access to the PMCs or their control structures from within the VM can be trapped to the hypervisor, the mechanism is additionally evasion-resistant (R2) and achieves a high level of stealth (R4). Due to these properties the proposed SILM approach is well suited for the use with security applications such as CRT-based detection of data-only malware.

7.4.1.2 Monitoring Specific Applications

The SILM monitoring mechanism presented in the last section enables us to selectively monitor instructions from the hypervisor. In general, however, we do not want to monitor the instructions of *all* applications, but rather of specific processes on the system. To accomplish this, we have to enable the PMCs whenever a process that we want to monitor is scheduled. In the following, we describe how application specific monitoring can be realized in the case of Linux guests.

First and foremost, we need to find a way to specify which applications we want to monitor and which we want to ignore. An intuitive way to do so would be to leverage the *name* of an application as identifier. This leads to the problem of how we can activate our monitoring mechanism whenever the application X that we want to monitor is started. To solve this problem, we intercept the execution of *every* new process and compare the name of the binary that is about to be executed with the list of applications we want to monitor. This can be achieved from the hypervisor by placing a hardware breakpoint on the function that is executed whenever a new process is created. In the case of Linux, this is the `execve` function, which receives the name of the binary that should be executed as an argument.

Having established that we want to monitor a specific process, the next issue that we have to address is how we can track the process across context switches. In particular, we have to disable monitoring whenever a different process is scheduled and we have to reenable monitoring whenever the monitored process is resumed. To achieve this, we can make use of a derivative approach and identify a process based on the `CR3` register value [67].

The `CR3` register contains the address of the page tables of a process. Since every process has its own address space, each process has a unique `CR3` register value. Whenever a process is scheduled, this value must be loaded into the `CR3` register. Since we can intercept moves to and from the `CR3` register from the hypervisor, we can use this property to detect when a process is scheduled. All that we require for this approach to work is the `CR3` value that belongs to a process that we want to monitor. Unfortunately, this value has not yet been computed on Linux when the `execve` function is invoked. This is why we intercept the execution of a second function, `start_thread`, to obtain the `CR3` register value of the process we want to monitor. Like `execve`, `start_thread` is executed for every newly created process.

Equipped with the `CR3` value of a process, we can activate our SILM mechanism whenever a process that we want to monitor is executed by the CPU. The only thing that is missing to complete our application specific monitoring mechanism is to identify when a process has finished its execution, as the `CR3` value of a process may be reused by another process once it exits. To accomplish this, we place a hardware breakpoint on the `exit` and `exit_group` functions, which are, as their name suggest, the counterpart to `execve` and `start_thread` and are executed whenever a process exits.

While the above describe approach is specific to Linux, we would like to stress that the techniques that we leverage are very generic. In fact, all that is required to implement a similar mechanism for another OS such as Windows, is to identify the equivalent functions that mark the beginning and the end of a process's execution. Since all other techniques (e.g. identifying processes based on their `CR3` register value) are based on hardware features, they are without modification directly applicable to other OSs as well.

7.4.1.3 Detecting ROP

Equipped with a secure SILM mechanism and a way to monitor specific applications, we can now present a concrete implementation of a CRT-based detection approach.

In particular, we want to detect ROP using the approach that we described at the beginning of this section. To refresh the reader, the idea behind this approach was to detect ROP based on `ret` instructions that have not been invoked by a corresponding `call` instruction. For this purpose we will implement a *shadow stack* on the hypervisor level: whenever a monitored application will execute a `call` instruction, we will push the address immediately following the `call` onto our shadow stack. On every `ret` instruction we then compare the address on top of the shadow stack with the new value of the IP. Since a function in general returns to the location immediately after the `call` instruction that invoked it, both values should be equal. In this case we will remove the address on top of the shadow stack such that we can validated the next `ret` instruction. Otherwise if the addresses should not match, the `ret` instructions does not have a corresponding `call` instruction and is thus likely a part of a ROP program.

However, there is a small caveat. While most benign `ret` instructions have a corresponding `call` instruction, there are a few exceptions. For instance, as we described in Section 2.2.1.4, the Linux kernel manipulates the stack when it delivers a signal to an application. In the process, the kernel, amongst other things, pushes a return address onto the stack that does not have a corresponding `call` instruction. To account for such rare cases, we make use of a *sliding ratio*. The idea thereby is to allow a certain amount of mispredicted `ret` instructions (i.e. `ret` instructions that do not have a corresponding `call` instruction) as long as their number does stay within a specific ratio to correctly predicted `ret` instructions. However, since the ratio is sliding, it does not refer to the absolute amount of executed `ret` instructions, but only to the recently executed instructions. For instance, a sliding ratio of 2/100 would state that there may occur two mispredicted `ret` instructions if the remaining 98 of the *last* 100 `ret` instructions have been predicted correctly.

Note that we do not leverage an absolute ratio, since applications typically have a large amount of correctly predicted `ret` instructions, while there are only a few mispredicted `ret` instructions. Consequently, an absolute ratio would enable an attacker to execute multiple mispredicted `ret` instructions in a row as long as she stays below the ratio. For long running applications, this will, due to the large amount of correctly predicted `ret` instructions, typically provide the attacker with the possibility to execute a significant amount of mispredicted `ret` instructions before she will be detected. This is illustrated in column four and five (Stats Normal) of Table 7.3, which shows the number of correctly predicted (+) and mispredicted (-) `ret` instructions for well-known Linux applications. As we can see, the mispredicted `ret` instructions are significantly lower than the correctly predicted `ret` instructions. If we would leverage an absolute ratio of one in a hundred (1/100) for example, an attacker would in the case of `ps` be able to execute 2,082 mispredicted `ret` instructions *in a row*, when the attack is performed at the end of the program's execution. A sliding ratio on the other side will always only allow the for the window configured number of mispredicted `ret` instructions independent of when the attack is performed. We will provide a more extensive discussion of the detection results of our approach later on in this section.

	Runtime [s]			*Overhead*	
Program	Normal	SILM	TF	SILM	TF
ls	0.001	0.014	0.035	**14.10**	**34.74**
ps	0.003	0.906	3.432	**359.71**	**1,362.04**
df	0.002	0.828	3.459	**413.79**	**1,729.54**
dig localhost	0.454	1.020	2.655	**2.19**	**5.85**
factor 121	0.001	0.009	0.054	**10.00**	**59.97**
cat /etc/passwd	0.001	0.007	0.028	**11.28**	**43.31**
sha1sum /etc/passwd	0.001	0.013	0.043	**12.54**	**43.18**
file /bin/ls	0.002	0.830	4.703	**415.01**	**2,351.69**
Sleep 1	1.002	1.027	2.296	**1.02**	**2.29**
id	0.489	0.614	1.234	**1.26**	**2.52**
Average	**1.45**	**1.92**	**5.05**	**112.91**	**405.14**

Table 7.2: Average performance results of our shadow stack when monitoring common Linux applications. Each program was thereby executed 100 times. The columns show from left to right: the executed program, the runtime in seconds of the normal execution of the program and when monitored with a shadow stack implemented via SILM and TF, and the average overhead of both implementations.

To evaluate the performance of the approach and to demonstrate the advantage of SILM, we implemented the above described shadow stack in two different ways. First we used an existing ILM approach, the TF, and trapped every instruction executed by the guest system. Whenever we encountered a `call`, we saved the address following the instruction to the shadow stack and on every `ret` we validated the target IP using the pushed addresses. In addition, we implemented the shadow stack using our SILM mechanism, which we set to *only* monitor `call` and `ret` instructions. We then set both implementations to monitor common Linux applications and recorded the overhead of both approaches. All experiments where thereby conducted on an Intel Core i7-4770 CPU with 3.4GHz and 16GB of memory. While we used Debian 7.6 with Linux kernel 3.14 as host OS, the guest OS was Arch Linux running kernel version 3.15.8. The results of this experiment are shown in Table 7.2.

As one can see, the SILM implementation is significantly faster than a traditional ILM approach. While our TF-based shadow stack decreased the performance of a monitored program by a factor of 2,352 in the worst case, the worst case performance decrease in case of SILM was 415. On average the SILM shadow stack incurred an overhead of 113 times in contrast to the TF-based approach which incurred an average overhead of 405 times. Consequently, the SILM shadow stack is about 3.5 times as fast as its traditional counterpart.

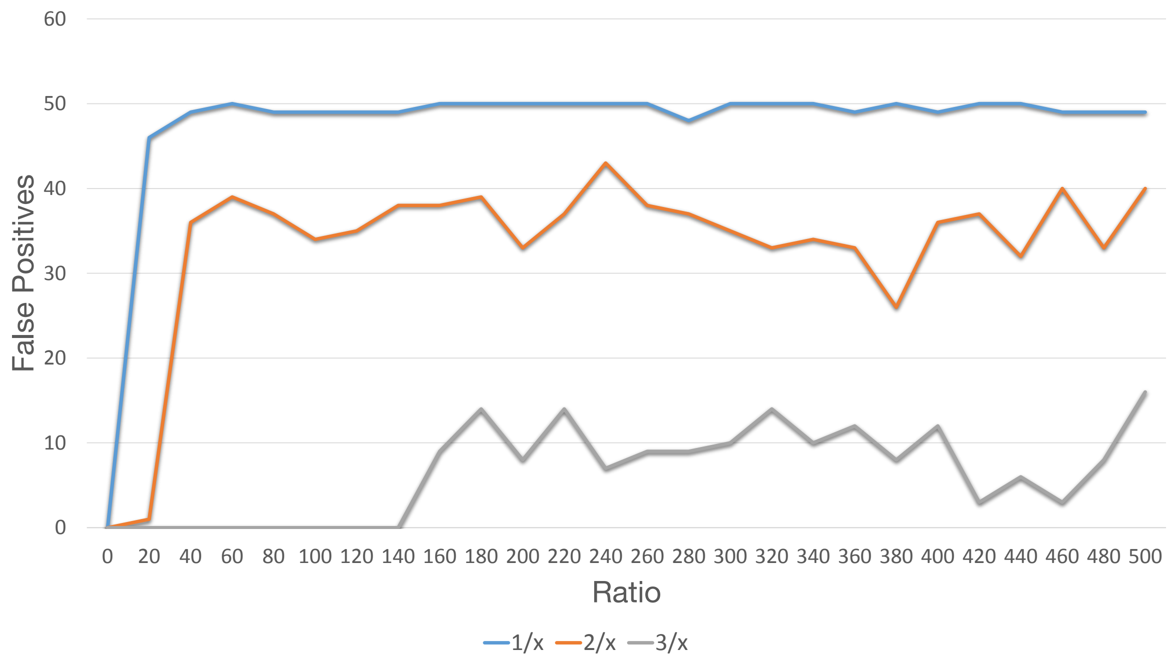


Figure 7.2: The effects of different sliding ratios on the shadow stack. In particular, the figure shows three different ratios 1/x, 2/x, and 3/x where x goes from 0 to 500 and the false positives for each ratio.

Even the SILM approach leads to a significant overhead though. We would like to stress, however, that while this overhead seems huge at first glance, it is not as devastating in practice as one would expect. In particular, the shadow stack incurs an especially large overhead when the runtime of the measured application is small (e.g. `ps`, `df`, or `file`), while its overhead decreases with the runtime of the monitored application (e.g `dig` or `id`). Consequently, the longer an application runs, the less noticeable will be the influence of the shadow stack. In addition, even for `file`, where our shadow stack has the highest overhead, the effective average runtime remains below a second. This is acceptable in many practical scenarios.

Having determined its performance, we then empirically evaluated the effectiveness of our approach against data-only malware. Since the detection capability of our shadow stack heavily depends on the sliding ratio that we use, we conducted two separate experiments for this purpose. To setup our shadow stack, we first determined the best setting for our sliding ratio. For this purpose, we executed the `dig` program multiple times within the VM and monitored its execution with our SILM shadow stack. In the process, we gradually increased the ratio in steps of 20 starting with a value of 1 until we reached the value of 500. For each ratio that we tested, we executed `dig` a hundred times and recorded the average number of false positives that we observed. The results of the experiment are shown in Figure 7.2.

If a ratio of 1/x is used, where x goes from 1 to 500, we see the number of false positives steeply rising. We reach fifty false positives when the ratio reaches 1/40. In this case essentially every second execution leads to a false positive as we executed the monitored program a hundred times. When we further increase the ratio, the number of false positives essentially stays the same.

In the beginning, the ratio of 2/x shows a similar development. In this case the number of false positives never reaches 50, however, but is oscillating at 40. The first time the ratio almost reaches 40 is at 2/60. From there on the ratio spikes in both directions. This implies that the number of false positives is only partly dependent on the size of the sliding window. Based on this observation, we can reason that the events which lead to false positives are not equally, but randomly distributed in the execution of the program. In case of 2/380, for instance, the number of false positives drops to 26 and is thus about 18 false positives lower than when a sliding ratio of 2/240 was used. If the false positives were equally distributed, however, we should actually see more false positives when a ratio of 2/380 is used as the window size is 140 `ret` instructions larger compared to 2/240.

The last ratio that we analyzed is 3/x. In contrast to the other ratios, we did not observe any false positives in this case before we reached a ratio of 3/140. From there on, however, their number increases noticeable. At 3/180 we can already observe 15 false positives. When we further increase the window size of the sliding ratio, the graph exhibits a similar development as the 2/x ratio and oscillates around 10 false positives. As before, we can observe unexpected drops in the process. At 3/460, for instance, the number of false positives drops to 4.

Since 3/x shows a stable development at the beginning, we chose to configure our shadow stack to use a ratio of 3/100. This provides a decent window size, while at the same time it leaves some room for outliers as we do not fully deplete the zero false positive range.

To verify the effectiveness of our approach against data-only malware, we then created infected versions of all the Linux applications shown in Table 7.3 using the file-based infection approach described in Section 3.5.2. In addition, to be able to test the applicability of our shadow stack to persistent data-only malware, we created a program (`persistent`) that periodically invokes a malicious data-only program and thus simulates the typical hooking behavior of persistent malware. We then monitored the execution of the normal and the infected program a hundred times and recorded the false positives and false negatives. The results of the experiments are shown in Table 7.3.

As we can see, the infected versions of the programs incurred a much higher number of mispredicted `ret` instructions as the uninfected versions, especially in the case of the persistent data-only malware, which did not execute once but multiple times. The shadow stack detected the malware in all cases reliably. Due to our experimentally obtained sliding ratio, we did not observe any false positives either. This shows that CRT-based detection can indeed be very effective against data-only malware in practice. Finally, note that while our shadow stack is primarily target at ROP, two of the three remaining

	Detection		*Stats Normal*		*Stats Infected*	
Program	FP	FN	+	-	+	-
ls /	0	0	3,077	**2**	19,552	**207**
ps	0	0	208,292	**4**	224,768	**209**
df	0	0	189,650	**3**	206,119	**208**
dig localhost	0	0	189,247	**5**	205,727	**211**
factor 121	0	0	1,736	**2**	18,208	**207**
cat /etc/passwd	0	0	1,348	**2**	17,829	**207**
sha1sum /etc/passwd	0	0	2,543	**2**	18,998	**207**
file /bin/ls	0	0	190,138	**3**	206,610	**208**
Sleep 1	0	0	1,312	**2**	17,793	**207**
id	0	0	142,635	**3**	203,563	**208**
persistent	0	0	3,228,679	**10**	3,290,115	**40,970**

Table 7.3: The table shows the detection results of our shadow stack when using a sliding ratio of 3/100. The columns show from left to right the executed program, the number of false positives (FP) and false negatives (FN) and the number of correctly predicted (+) and mispredicted (-) `ret` instructions in case of the normal and the infected program.

code techniques discussed in Section 2.2 make also use of `ret` instructions to function. These code reuse techniques could thus similarly be detected by our approach.

7.4.1.4 Advantages & Disadvantages

As demonstrated in the last section, CRT-based detection can be very reliable. In addition, CRT-based approaches can, once a suitable ILM mechanism is available, often be easily implemented. This is also due to the fact that CRT-based approaches in general do not require additional information such as debug information or source code to function. In addition, CRT-based detection mechanisms can be applied to existing applications without changing or recompiling them, which makes them easily deployable.

However, CRT-based detection also has some important disadvantages. First, every code reuse technique may require its own detection mechanism. While the approach presented above is capable of detecting ret2libc, ROP, and SROP, it is unable to detect JOP. Consequently, a CRT-based detection approach is in general not universal.

Second, even though most code reuse techniques currently operate on the instruction-level, future techniques are in no way bound to this convention. SROP, for instance, primarily relies on a software construct to function. While it still leverages the `ret` instruction to combine instruction sequences, the approach shows that it might be possible to create code reuse techniques that solely operate on the software-level. Detecting such techniques merely based on the executed instructions will be quite difficult.

Third, CRT-based detection is only able to detect *known* code reuse techniques. Novel techniques will, however, go unnoticed until a detection mechanism is created. Thus CRT-based detection faces a similar limitation than signature-based detection. Note, however, that it is much more difficult to create a new code reuse technique compared to changing a signature. For instance, after the introduction of ret2libc it took almost 10 years till ROP was presented.

Fourth, as our implementation has shown, CRT-based detection techniques in general require heuristics to avoid false positives. These heuristics can be leveraged by an attacker to evade the mechanism. In our implementation, for example, an attacker can execute three malicious `ret` instructions every 97 benign `ret` instructions. This is, however, only the case if there are no benign mispredicted instructions while the attack is performed. Consequently, to reliably evade the mechanism, the attacker must actually be able to predict when benign mispredicted `ret` instructions will occur. Since our experiments have shown that these instructions appear randomly during execution, however, this is a hard problem in practice. Thus while our shadow stack provides a small attack surface in theory, reliably evading the mechanism is not straightforward. To further reduce the chances of a successful attack, one could in addition to the sliding ratio also count the absolute number of mispredicted `ret` instructions. As our experiments have shown, this number should usually be quite small.

Fifth and finally, ILM even SILM leads to a noticeable performance overhead.

7.4.1.5 Related Work

Due to their reliability researchers proposed various different CRT-based detection approaches. The majority of them, however, is targeted at the detection of code reuse exploits instead of data-only malware. Most closely related to our work is ROPdefender [39], which was proposed by Davi, Sadeghi, and Winandy and also leverages a shadow stack to detect ROP. In contrast to the approach presented within this thesis, the mechanism, however, relies on instrumentation to obtain all executed `call` and `ret` instructions and can thus not be used in a virtualized environment. Instead, ROPdefender leverages the Pin framework[2]. To use this approach, however, ROPdefender must execute at the same privilege level as the monitored application. Consequently, the approach is only suited for prevention, but not for the detection of data-only malware as malware could directly attack the mechanism and disable it.

An interesting alternative to implementing an entire shadow stack has been presented by Wicherski [178]. Instead of leveraging the PMCs to trap events, Wicherski proposed to use them directly for ROP detection. To achieve this the counters are set to raise an interrupt whenever *mispredicted* `ret` instructions are encountered. Since x86 processors internally manage a shadow stack to predict the location of `ret` instructions, this has, in theory, the same effect as using an own shadow stack. However, the approach unfortunately

[2]https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

leads to false positives in practice (we assume this is due to the fact that the space of the hardware stack is limited). To counteract these false positives the author proposes to check whether a `ret` instruction returns to a `call` preceded instruction. As discussed in Section 4.4.3, this approach can be easily circumvented though.

Another common approach to CRT-based detection is to measure the frequency of `ret` instructions (e.g. [23, 38]). The idea behind this approach is that gadgets are usually quite short. Thus `ret` instructions will occur frequently when ROP is used. During normal operation, however, `ret` instructions only occur after the execution of a function. Therefore the ratio between `ret` instructions and other instructions will be much higher when ROP is leveraged compared to normal instruction execution. However, to evade such an approach the attacker only needs to leverage longer gadgets. In its simplest form, this can be accomplish by using ret2libc.

Besides detecting the code reuse technique, it is also possible to *remove* all gadgets from executable code by recompiling existing binaries and removing all instructions that the attacker could leverage to perform code reuse attacks. Li et al. [82] present such an approach and essentially replaces both `call` and `ret` instructions with `jmp` instructions to hinder attackers from performing ROP. Another compiler based approach to mitigate ROP was introduced by Onarlioglu et al. [104]. They introduce alignment sleds between different instructions, which enables them to also prevent unintended `ret` or `jmp` instructions. In addition, they also encrypt addresses used for returns or jumps by XOR-ing random data with the addresses. As a consequence, intended gadgets cannot be used by an attacker either. Since these approaches rely on the compiler, however, they require the recompilation of all existing binaries and cannot be directly applied to existing applications.

Finally, CFI mechanisms also often leverage rules that target the code reuse technique. Since we discussed CFI in detail in Section 4.4.3, however, we will not consider such approaches here any further.

7.4.1.6 Summary

In this chapter we proposed the concept of CRT-based detection. The idea behind the approach is to detect the code reuse technique that the data-only malware leverages instead of detecting the malware itself. For this purpose we first introduced a general ILM mechanism that enables security applications to selectively monitor instructions within a guest system from the hypervisor in a evasion-resistant and stealthy manner. With the help of this mechanism we then presented a shadow stack implementation as an example of a CRT-based detection mechanism for ROP. The experiments that we conducted show that CRT-based detection can be very effective and can be directly leveraged to protect existing applications. However, CRT-based detection incurs a significant overhead, requires heuristics, and is on top of that code reuse technique specific. As a consequence, the approach is only able to detect known code reuse techniques and is vulnerable to evasion attacks.

7.4.2 Detecting the Control Structure

As discussed in Chapter 4, an effective countermeasure against traditional malware is to detect the malware based on its code region. The equivalent of a code section in the case of data-only malware is the control structure. Consequently, one would assume that the control structure represents a natural basis for a detection mechanism. In spite of this fact, research so far primarily focused on other properties of data-only exploitation for its detection. For instance, many researchers attempt to identify whether a certain instruction sequence is used as a gadget (e.g. [27, 107, 185]). The control structure on the other side has hardly attracted any attention, which is surprising given that it is one of the fundamental dependencies of data-only malware. In this section, we make up for this lapse by presenting a detection approach that aims to identify data-only malware based on its control structure. To provide a general mechanism, we will attempt to leverage the fundamental properties of control structures for their detection. However, to realize such an approach, we have to identify these properties first.

7.4.2.1 The Properties of Control Structures

P1 **Memory-Persistence.** Similar to the code of traditional malware, a control structure must be residing in memory as long as the data-only malware is executing.

P2 **Pointer-Layout.** Independent of the code reuse technique that is used, control structures consist of pointers to instructions. While an attacker could try to hide these pointers by encrypting or obfuscating the control structure, there must, as in the case of traditional malware, exist a decryption sequence whose pointers cannot be hidden. Consequently, a control structure must at least contain a single code pointer. However, in practice control structures are likely to contain dozens of such pointers.

P3 **Size.** Control structures generally require a lot more space than executable code. This is especially true for control structures of data-only malware which are usually huge as our implementations of data-only malware (see Section 3.5 for details) have shown.

7.4.2.2 A General Approach for the Detection of Control Structures

Based on these properties, we will in the following present a general approach for the detection of control structures and thus for the detection of data-only malware. The fundamental idea behind this approach is to detect control structures based on their structure (P2) and size (P3), as we believe that these properties can be leveraged to distinguish normal control data of the application from control structures. More precisely, we consider control structures to be an *anomaly* within memory (P1) that can be detected. On a high-level the anomaly-based detection approach that we propose attempts to detect control structures within memory (P1) by finding pointers into code regions (code pointer) and trying to determine whether the pointers belong to a control structure or not. The detection approach thus leverages the pointer-layout (P2) of a control structure for its detection. The size (P3) of the control structure will thereby increase the reliability of the approach as the number of malicious code pointers should increase with the size of the control structure.

To implement such an approach, we must consider three fundamental questions: First of all, how can we identify code pointers within memory, which is a requirement for the detection approach? Second, how can we distinguish benign control data from a control structure? And third and finally, when should we perform the validation such that the mechanism achieves a good performance and a high reliability? In the following we will consider each of this questions in turn.

Identifying Code Pointers. At closer inspection the identification of code pointers is actually quite straightforward. To detect code pointers, we can simply iterate through the memory on a byte granularity, cast the bytes at the current position into a pointer, and check whether the resulting pointer points to a code region. However, for this approach to work, we first have to identify all code regions of the system. This can be accomplished with the help of the page tables.

To be able to contain code, a memory region must be marked as executable. On the x86 platform this is exactly then the case when a memory page has its execute-disable (XD)[3] cleared within the page tables [64]. Consequently, we can obtain all executable pages by iterating through the page tables[4] and extracting every page that has its XD cleared. In the process, we can also obtain all mapped memory pages, which enables us to limit our search space to the memory regions that are actually used by the application thus increasing the performance. Moreover, based on the supervisor-bit of each page table entry we can even distinguish between kernel space and user space, which is helpful if we only want to consider kernel memory or user memory. Finally, since we leverage the page tables to obtain this information, the mechanism is *binding*.

Validating Code Pointers. Once we obtained all pointers to code regions, we have to distinguish between benign code pointers and code pointers that belong to control structures. To achieve this we leverage multiple heuristics. In the process we differentiate between two fundamental cases: either the control structure is located on the stack or the control structure resides somewhere else in memory.

[3]The execute-disable (XD)-bit provides hardware support for the realization of W $\oplus$ X. As the name suggests, the processor will generate an exception whenever an attempt is made to execute an instruction from a memory page that has its XD-bit set within the page tables.

[4]As explained in the last section, the address of the page tables is stored within the `CR3` register and can thus be obtained from there.

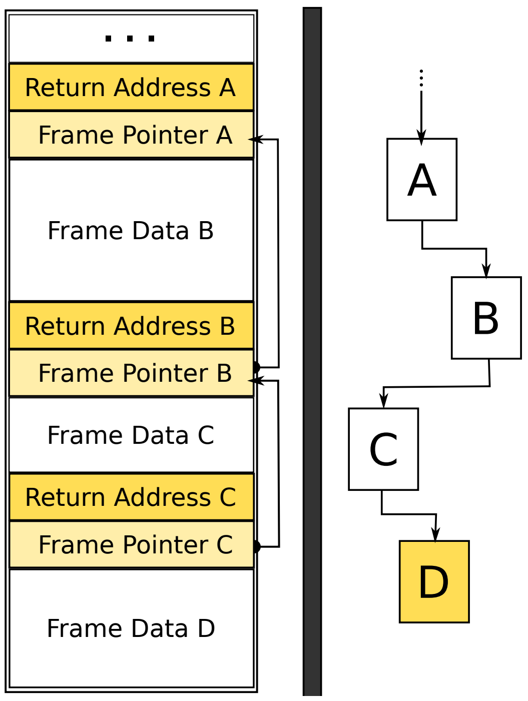


Figure 7.3: The figure shows the relation between the CFG of a program (right) and its stack (left) and thus visualizes the heuristics that have been discussed in Section 7.4.2.2. The currently executing function is D.

Control Structure Residing on the Stack. The stack has a very specific layout. In particular, it consists of individual function frames. Each frame thereby stores the return address of the function as well as its local variables. In addition, there exists a connection between the individual function frames: given the CFG of the application, there must exist a path that connects all frames. This is due to the fact that a frame essentially represents a function. Consequently, if a function frame A directly follows another frame B, it implies that function A must have invoked by function B.

Control structures on the other side have a very different layout. In particular, there are neither function frames nor is there by default a connection between the individual instruction sequences that are reused. To identify control structures on the stack, we can thus extract each function frame from the stack and check whether the frame corresponds to the specific format we described above. For this purpose we will leverage the following heuristics:

Return Addresses Every code pointer on the stack that does not point to the beginning of a function is a return address.

Path Each return address marks the beginning of a function frame. There must be a path in the CFG of the application that leads from one return address to the next starting from the first frame in the stack, which is the frame with the highest address.

Frame Size Each function frame has a specific size that depends on the function it belongs to.

Base Pointer If a function has a base pointer, this base pointer must point to the field before the previous return address.

These heuristics are visualized in Figure 7.3. To validate them, we require a CFG of the application to monitor, the address of each function within the application as well as its frame size, and the information whether it has a base pointer. All of this information can in general be extracted from an executable.

Control Structure not Located on the Stack. If a benign code pointer lies outside of the stack region[5], it generally is a function pointer. To distinguish between benign function pointers and pointers within a control structure, we essentially require all locations where the application stores benign function pointers and the functions they can point to. This exactly corresponds to the functionality that current hook-based detection mechanisms provide. Consequently, we can leverage these mechanisms to obtain this information. The validation of the code pointers itself can then be performed by checking whether each pointer is stored at one of the identified function pointer locations and if it points to a valid target. Each code pointer that does not fulfill this criteria is part of a control structure.

Performing the Validation. The final question that we must address is when to actually perform the validation. There essentially are two approaches that can be leveraged: we can either perform the validation constantly in a loop or whenever a specific event occurs. In the former case, we will start a new validation as soon as the current validation finishes. Since we can perform the validation from the hypervisor and do only need to interrupt the execution of the guest to obtain the memory areas we require for the validation, this will only slightly increase the runtime of the monitored program and increase the likelihood of detecting an attack. However, we will also use an entire CPU for the validation, which will decrease the overall performance of the system. Even worse, for each program that we monitor we will require an own CPU.

Alternatively, we can only perform the validation whenever a specific event occurs. Most interesting in this case would be to start the validation whenever we think there might be an attack going on, but we are not sure yet. To accomplish this, we could combine the mechanism with other detection mechanisms to increase their accuracy. For instance, could we leverage the shadow stack approach that has been presented by

[5]Whether a code pointer lies outside of the stack region can be determined by obtaining the value of the SP and checking whether the pointer lies outside the area SP + maximal stack size. We will discuss this approach in more detail in Section 7.4.3.

Wicherski [178] and perform the validation whenever an unpredicted `ret` instruction occurs.

7.4.2.3 Validating the Stack

Since current hook detection mechanisms are already able to detect and validate code pointers that lie outside of the stack region, the main problem that remains to implement a detection mechanism for control structures is the validation of the stack. In the following we will discuss a possible approach to solve this problem and evaluate its effectiveness against data-only malware.

Implementation. In accordance with our system defense model, we implemented a virtualization-based system that is capable of validating the integrity of the stack of an application, which is running within a guest system, from the hypervisor. Once more we leveraged X-TIER as a secure foundation for our implementation. To perform the validation itself, our system relies on the heuristics described in the previous section. For this purpose the system supports two different modes of operation: *analysis* mode and *validation* mode.

To be able to leverage the identified heuristics, our system first requires detailed information about each of the applications that should be monitored. The system obtains this information in analysis mode. In particular, it will generate a CFG for each binary that should be protected and extract details about each function in the program such as its frame size and whether it leverages a frame pointer. In the process, the system will also consider any library that the application loads. Currently our implementation only supports Linux ELF applications. However, because most of the analysis code operates on the assembly level, it could be easily extended to support other OS as well.

Once the necessary information has been obtained, the system can be switched to validation mode. In validation mode, the analyzed applications are monitored and the actual integrity validation of the stack occurs. Whenever a validation is triggered (we will describe the events that lead to a validation in more detail in the next paragraph) the system first attempts to identify each of the individual function frames on the stack starting from the current position of the SP, which marks the top of the stack. This process is also referred to as "unwinding" the stack. Our system relies on `libunwind`[6] for this purpose.

Once a frame has been identified by `libunwind`, its size and frame pointer are validated. In addition, it is ensured that there exists a path between the current function frame and the frame that resides directly below[7] it on the stack. Only if all three of these conditions

[6]`Libunwind` is a popular open source library that was specifically designed for the task of unwinding the stack. It is for instance used by the Linux kernel. The interested reader can find more information about `libunwind` at http://www.nongnu.org/libunwind/.

[7]Since the stack grows downwards on the x86 architecture, this function frame will have a *higher* address than the current frame.

	Runtime [s]				
Program	Normal	1	10	100	1000
ls	0.001	0.127	0.028	0.006	0.003
ps	0.003	0.749	0.149	0.036	0.016
df	0.002	0.671	0.130	0.031	0.012
dig localhost	0.012	0.413	0.120	0.033	0.017
factor 121	0.001	0.129	0.027	0.005	0.003
cat /etc/passwd	0.001	0.131	0.026	0.005	0.003
sha1sum /etc/passwd	0.001	0.126	0.029	0.006	0.003
file /bin/ls	0.002	0.645	0.129	0.032	0.013
sleep 1	1.002	1.131	1.026	1.006	1.003
id	0.002	0.488	0.099	0.024	0.011
Average	**0.1027**	**0.4610**	**0.1763**	**0.1184**	**0.1084**
Overhead	**0**	**2.630**	**0.388**	**0.153**	**0.055**

Table 7.4: Performance results of our approach to validate the integrity of the stack when monitoring common Linux applications. Each program was thereby executed 100 times. The columns show from left to right: the executed program, the runtime in seconds of the normal execution of the program without monitoring as well as when monitoring every, every tenth, every hundredth, and every thousandth `return` instruction.

are met, will our system consider the frame to be valid. If each function frame on the stack is valid, the stack is considered to be with integrity.

Experiments. To measure the performance overhead of our approach, we conducted four different sets of experiments. In particular, we set our system to validate the stack of a monitored application on every `return` instruction, on every tenth `return` instruction, on every hundredth `return` instruction, and on every thousandth `return` instruction. To accomplish this we once more leveraged our SILM mechanism (see Section 7.4.1.1 for details). We chose this event-based validation approach instead of a constant validation approach for two reasons: first of all, since constant validation requires an individual CPU for each monitored application, the approach simply does not scale in practice. Second, by only validating the stack when specific events occur, our system will only be able to observe a small part of the application's execution. This makes it much more challenging to detect attacks, which enables us to evaluate the effectiveness of our system under realistic conditions. The `return` instruction thereby naturally seemed as a good candidate to trigger the validation, since it obtains its control data directly from the stack. However, the validation could similarly performed based on any other event.

The results of the performance experiments are shown in Table 7.4. All experiments where performed on an Intel Core i7-4770 CPU with 3.4GHz and 16GB of memory. We used Debian 7.6 with Linux kernel 3.14 as host OS, while the guest ran Arch Linux with kernel version 3.16.2. To allow for an easy comparison between the approaches, we chose to monitor the same set of Linux applications that we used for our previous experiments. Each application was thereby executed a hundred times.

As expected, the performance of the approach improves the less `return` instructions we trap. When every `return` instruction is trapped, the monitored program runs on average about four times slower compared to when the program is executed without monitoring. However, it must be mentioned that the overhead of some programs deviates considerably from this average value. For example, when we monitor every `return` instruction, the monitored `file` program is about 322 times slower compared to its normal execution.

The overhead of the monitoring drops significantly when we only monitor every tenth, hundredth, or thousandth `return` instruction. For every tenth `return` instruction the overhead already reduces to about 0.4 on average, while it reduces to 0.2 in the case of trapping every hundredth `return` instruction and to 0.06 when only trapping every thousandth `return` instruction. However, this increase in the performance comes at the cost of security. The less instructions we trap, the less validations we perform, which makes it more likely that we miss an attack as we will show in the following when we discuss the effectiveness of our approach.

To evaluate the effectiveness of our approach against data-only malware, we used our file-based infection mechanism that we presented in Section 3.5.2 to infect each of our test applications. We then counted the number of stack validations that were performed for both the uninfected and the infected version of each binary when trapping every, every tenth, every hundredth, and every thousandth `return` instruction. In the process, we also counted the number of successful validations (i.e. stacks whose integrity could be validated) for each version of the binary. For this purpose, we once more used a sliding window of one hundred `return` instructions. That is, we counted the number of malicious stacks that occurred in a window of one hundred validations. As explained in the previous section, this approach guarantees that we are able to detect attacks on long running applications as well, which we might miss if we would take the absolute ratio instead. To test our approach against persistent data-only malware, we repeated this experiment with the `persistent` program. Each program was executed a hundred times. The average results are shown in Table 7.5 and Table 7.6.

As can be seen, there is, with the exception of `persistent`, a significant difference in the number of malicious stacks per sliding window for the infected and uninfected version of each tested program. While the maximal number of bad stacks in all windows of all uninfected programs was 7 (`dig` and `ps`), most of the infected programs had at least a single window that entirely consisted of bad stacks. In addition, we can observe a clear discrepancy between the average number of bad stacks per window for most of the tested programs. For example, in the case of `ls` only about one stack (0.86) in each window was found to be malicious when every `return` instruction of the uninfected program was

	Validations (B)			Validations (I)		
PROGRAM (1)	T	MAX	∅	T	MAX	∅
ls	1,862	5	0.86	18,488	100	89.89
ps	172,742	7	0.04	190,899	100	8.73
df	156,511	3	0.01	172,723	100	9.61
dig localhost	104,654	7	0.05	115,763	100	14.38
factor 121	1,193	5	0.92	17,804	100	93.28
cat /etc/passwd	1,004	3	0.60	17,567	100	94.51
sha1sum /etc/passwd	1,738	6	0.81	18,324	100	90.67
file /bin/ls	153,097	4	0.01	166,297	100	9.99
sleep 1	939	3	0.64	17,561	100	94.56
id	114,543	3	0.01	131,513	100	12.63
persistent	931,917	3	0.00	1,020,976	24	8.40
PROGRAM (10)	T	MAX	∅	T	MAX	∅
ls	229	3	1.75	1,894	100	87.91
ps	19,915	4	0.05	21,433	100	7.78
df	17,925	1	0.01	19,394	100	8.58
dig localhost	16,457	5	0.07	17,971	100	9.30
factor 121	155	2	1.94	1,815	100	91.57
cat /etc/passwd	120	2	1.67	1,780	100	93.37
sha1sum /etc/passwd	211	4	2.37	1,874	100	88.85
file /bin/ls	17,764	4	0.04	19,570	100	8.52
sleep 1	114	1	0.88	1,775	100	93.58
id	13,375	1	0.02	15,041	100	11.06
persistent	262,378	1	0.00	261,481	4	2.94

Table 7.5: The first part of the results of the experiments that we conducted to validate our control structure-based detection approach. We conducted four sets of experiments. In each set we reduced the number of `ret` instructions that were trapped. While we trapped every `ret` instruction in the first set, we trapped only every tenth, every hundredth, and every thousandth `ret` instruction in the second, third set, and fourth set respectively. In each set, each program was thereby executed 100 times. The columns show from left to right: the executed program, the total number of stack validations, the maximum number of bad stacks in a sliding window of one hundred `return` instructions, and the average number of bad stacks in all sliding windows of one hundred `return` instructions for both the benign (B) and the infected (I) program.

	Validations (B)			*Validations (I)*		
PROGRAM (100)	T	MAX	∅	T	MAX	∅
ls	22	0	0.0	189	100	87.83
ps	2057	0	0.0	2222	100	7.52
df	1823	0	0.0	1987	100	8.40
dig localhost	1843	0	0.0	2011	100	8.35
factor 121	12	0	0.0	180	100	92.22
cat /etc/passwd	10	0	0.0	176	100	94.32
sha1sum /etc/passwd	20	1	5.0	187	100	88.77
file /bin/ls	1823	0	0.0	1992	100	8.43
sleep 1	11	0	0.0	176	100	94.32
id	1351	0	0.0	1521	100	10.91
persistent	26145	0	0.0	28647	0	0.0
PROGRAM (1000)	T	MAX	∅	T	MAX	∅
ls	1	0	0.0	17	15	15
ps	203	0	0.0	219	16	7.31
df	178	0	0.0	197	16	8.12
dig localhost	182	0	0.0	198	15	8.08
factor 121	0	0	0.0	16	16	16
cat /etc/passwd	0	0	0.0	16	16	16
sha1sum /etc/passwd	0	0	0.0	17	15	15
file /bin/ls	180	0	0.0	196	16	8.16
sleep 1	0	0	0.0	16	16	16
id	134	0	0.0	150	16	10.67
persistent	2517	0	0.0	2768	0	0.0

Table 7.6: The second part of the results of the experiments that we conducted to validate our control structure-based detection approach. We conducted four sets of experiments. In each set we reduced the number of `ret` instructions that were trapped. While we trapped every `ret` instruction in the first set, we trapped only every tenth, every hundredth, and every thousandth `ret` instruction in the second, third set, and fourth set respectively. In each set, each program was thereby executed 100 times. The columns show from left to right: the executed program, the total number of stack validations, the maximum number of bad stacks in a sliding window of one hundred `return` instructions, and the average number of bad stacks in all sliding windows of one hundred `return` instructions for both the benign (B) and the infected (I) program.

trapped, while the infected program showed 90 (89.89) malicious stacks in each window in the same experiment. A difference of 89 frames per window on average.

Not surprisingly, the accuracy of the method reduces with the number of `return` instructions that we trap. If we once more take `ls` as an example, we can see that when trapping every thousandth `return` instruction, we did not observe any invalid stacks in case of the uninfected program, while we saw 15 (94.12%) malicious stacks for the infected program. At the same time, the number of validations decreased significantly. While the system validated 1,862 stacks (uninfected) and 18,488 stacks (infected) respectively when trapping every `return` instruction, it only validated one stack (uninfected) and 17 stacks (infected) respectively when set to trap every thousandth `return` instruction. Some of the normal executions of the tested programs (e.g. `factor`) even yielded no validations at all when just trapping every thousandth `return` instruction. Considering these observations for all tested programs, we find that our mechanism provides the best tradeoff between security and performance if we trap every hundredth `return` instruction. In this case most infected programs still show at least a single window that entirely consists of infected stacks, while the performance of the approach increases considerably.

An exception to this rule is the `persistent` program. Compared to the other programs, the detection results for this program are much worse. While there is still a notable difference between the benign and the infected version of the program when trapping every `return` instruction, this difference is no longer existent if we only trap every tenth, hundredth, or thousandth `return` instruction. The main reason for this is that the control structure that we use for `persistent` is very short. As a consequence, we miss the execution of the data-only malware as soon as we slightly reduce the number of events that we trap. This shows that, while being effective in most cases, the proposed mechanism will probably fail for short control structures in practice.

Lastly, note that the number of validations significantly differs between the uninfected and infected version of each program. This is a result of the fact that the infected version adds additional functionality to the benign program, the malicious code, that leads to the execution of additional `return` instructions. However, not all of these additional instructions must necessarily lead to malicious stacks. For instance, if the attacker uses ret2libc to call an existing function, the function will execute a valid path within the system. While the initial stack that diverts the control flow will be malicious, all other stacks that are part of the new execution path may be valid.

7.4.2.4 Advantages & Disadvantages

The most important advantage of the presented approach is that it is not focused on a particular code reuse technique, but is capable of detecting control structures in general. As a result, the approach is applicable to all forms of data-only malware independent of the code reuse technique they were implemented in. In addition, because it is, to the best of our knowledge, the first approach that attempts to detect data-only malware based on its control structure, attackers are currently unprepared for it, which increases the probability that the approach will be effective in practice.

However, the experiments conducted in the last chapter also illustrate the main weaknesses of the approach: it is based on heuristics. As a consequence, our system is currently unable to validate all stacks even if a program is benign. We assume that this problem is a result of abnormal system behavior that is not considered by our heuristics. For instance, as described in Section 2.2.1.4 the kernel may manipulate the stack during signal delivery by pushing an additional frame. Since such signal frames do not fulfill our definition of a valid stack, the validation of such frames will fail. We assume that there exist various of such boundary conditions in practice that must be considered before the mechanism can be applied to real world scenarios.

Second, if the executed data-only malware only consists of a few gadgets, we may miss the attacks if we only trap a part of the events. While this is true for all detection mechanisms, the proposed mechanism even aggravates the problem as the few malicious events that we observe may not all be recognized as such by our system due to its heuristics (false negatives).

Finally, since the mechanism is based on heuristics, it is in general vulnerable to evasion attacks. For instance, could an attacker design her control structure exactly in such a way that it simulates a valid stack layout. While this places a lot more constraints on the control structure, such an approach is in theory possible.

7.4.2.5 Summary

In this section, we presented an approach that detects data-only malware based on its control structure. Since the control structure is an integral part of any data-only program, leveraging it for a detection technique can result in an effective detection mechanism. The accuracy of the mechanisms will thereby depend on the quality of the heuristics that we leverage and the number of events that we trap. Since we rely on heuristics to perform the validation, however, the approach is in general not evasion-resistant.

7.4.3 Detecting the Switching Mechanism

Once the control structure of data-only malware has been loaded into memory, it must be *activated* to start the actual execution of the malware. This step is comparable to placing the IP to the code region of traditional malware, which leads to its execution by the CPU. In case of data-only malware, the code region of the malware is its control structure, while the IP is dependent on the code reuse technique that we leverage. For instance, to start the execution of a ROP program, we first need to place the SP, which serves as the virtual IP of the data-only program, to the beginning of our chain. In the next step, we then have to trigger the usage of the virtual IP using an *activating* instruction sequence. In the case of ROP, this can, for example, be accomplished with a `ret` instruction. Both of these steps form the *switching sequence* of the data-only malware.

Since the switching sequence is the third dependency that data-only malware has, it is like the control structure of data-only malware predestined for the creation of a detection mechanism. The detection of the switching sequence can thereby be conducted on two levels. On the one hand we can monitor the execution of the system (e.g. using SILM) and try to identify switching sequences when they are executed. However, this leads to the problem that switching sequences could also legitimately occur within code and we thus would have to distinguish between malicious and benign switching sequences. In addition, similar to signature-based detection, the approach would only allow us to detect known switching sequences, as we can only search for something that we know.

On the other hand, instead of focusing on the instruction sequences that perform the switch, we can alternatively try to detect the *effects* of a switching sequence. For example, switching sequences used with ROP commonly load the address of the control structure into the SP. Consequently, we can detect such switching mechanisms by monitoring the SP and watching for abnormal changes applied to it. For instance, can we check whether the SP is suddenly pointed to a completely different memory region, which is unusual behavior for many applications. Since such an approach would focus on the *behavior* of a switching sequence, it is in theory able to also detect unknown switching sequences. Depending on the heuristics that we apply to identify abnormal behavior, the approach may, however, suffer from a high false positive rate.

Since we are within this thesis in general more interested in detection mechanisms that have a wide applicability, we will in the following primarily consider the latter approach and experimentally evaluate its strengths and weaknesses.

7.4.3.1 Detecting Switching Sequences based on the Stack Pointer

Since three of the four code reuse techniques discussed within this thesis leverage the stack to control the execution of a data-only program, we implemented a simple detection mechanism that aims to detect switching sequences based on modifications of the SP. The idea behind this approach is to obtain the initial value of the SP when a program is launched and to validate that the SP always remains within a certain boundary. In particular, given that the program does not legitimately change its stack, which is a very uncommon operation, the SP should always be smaller (the stack grows down) or equal as its initial value. The maximal amount that the SP can differ from its initial value depends on the maximal stack size that a process can have. On Linux the maximal stack size of a process is in general fixed to 8,192 kB. Consequently, the SP should always remain in the range of:

$$SP <= initial\_sp\_value \;\&\&\; SP >= initial\_sp\_value - (8192 \cdot 1024)$$

Based on this formula we implemented a detection tool on top of our SILM mechanism and X-TIER for Linux. In particular, we set our SILM mechanism to execute a validation function whenever the monitored program executes a certain number of `ret` instructions.

The validation function would then obtain the current value of the SP and validate its value using the formula stated above. The initial stack value was thereby obtained by trapping the `start_thread` function using a hardware breakpoint (see Section 7.4.1.2 for details). Since the `start_thread` function gets the initial SP value of a process as argument, this can simply be achieved by reading the corresponding argument when the function is invoked.

As stated above, we did not execute our validation function on every return. The reason for this is that the suggested approach does, in contrast to our shadow stack implementation, not require every return value to function. This enables us to improve the performance of the approach by reducing the number of `ret` instructions that are trapped. This functionality comes at a cost though: the less instructions we trap the more seldom will our validation function be executed. Consequently, by increasing the performance of the approach, we also decrease its security.

To evaluate this trade off between security and performance, we conducted two different experiments. To measure the performance of the approach, we first once more monitored the execution of the same Linux applications we used in our previous experiments. As before, each application was thereby executed a hundred times once with instrumentation enabled and once with instrumentation disabled. During the performance measurements we also recorded all false positives that we observed.

In the second experiment we then evaluated the effectiveness of our approach by infecting each of the applications using the file-based infection approach presented in Section 3.5.2. To test the applicability of the approach against persistent data-only malware, we also once more made use of our `persistent` program to simulate the behavior of persistent data-only malware. In the process, we recorded all false negatives that occurred.

Each experiment thereby in turn consisted of three different sets of tests. During each set we reduced the number of trapped instructions and recorded the effects. In the first step, we set our SILM mechanism to trap every tenth `return` instruction. In the second set, we increased this number to a hundred, while we only trapped every thousandth `return` instruction in the last set. All experiments were thereby once more performed on a Intel Core i7-4770 CPU with 3.4GHz and 16GB of memory. As before, Debian 7.6 with Linux kernel 3.14 was used as host OS, while the guest ran Arch Linux with kernel version 3.15.8. The results of these experiments are shown in Table 7.7.

As expected the overhead of the approach decreases with the number of traps. For example, while the `file` program suffered from an overhead of factor 23 when we monitored every tenth `return` instruction, this overhead reduced to a factor of 1 when we only monitored every thousandth `return` instruction. However, at the same time, the security of the mechanism reduced significantly. This can best be seen based on the `persistent` program. While we detected 8,211 stack switches when we trapped every tenth `return`, we only detected a single stack switch when we trapped every thousandth `return`. In addition, we observed 8 false negatives, which clearly shows that we will miss more and more switches the less we trap. Trapping every hundredth `return` instruction thereby seems to provide a good trade-off between security and performance. In this case we still reliably detect all infections without false positives or false negatives and at the same time reduce the overhead from a factor of 23.11 to a factor of 3.51 in the worst case. All in all our results show that switch based detection can provide good detection results with good performance.

While we focus on the detection of a stack switch within this thesis, similar approaches can also be leveraged to detect non-stack based switching mechanisms. To detect JOP, for instance, we could monitor *all* registers of an application and validate whether a single general purpose registers always points to the same instruction sequence. While this will not be true for most normal applications, JOP requires one register to point to the dispatcher gadget, which is an integral part of the switching mechanism of a JOP program. Consequently, this register will be constant.

Finally, we selected the `ret` instruction for our implementation, since it accesses the stack to obtain the next instruction that should be executed. Due to this property it is naturally a good candidate to detect stack switches. The approach could, however, also be implemented by trapping any other event. It is not restricted to the `ret` instruction alone.

7.4.3.2 Advantages & Disadvantages

As the experiments presented in the last section show, the proposed approach only occurs a small overhead and is on top of that very effective and reliable. In addition, the mechanism can be easily implemented and does not require any extensive knowledge about the monitored application. Consequently, it can be easily deployed to protect existing systems. The simplicity of the mechanism also leads to two significant disadvantages though. First and foremost, the mechanism is unable to detect data-only programs that are directly written onto the stack as the SP will not change significantly in this case. While this attack could potentially be detected by monitoring and validating instructions sequences that could be abused for a switching mechanisms, these sequences will in this case only consist of `add rsp` or `sub rsp` instructions which frequently occur within code and are thus difficult to verify. In the worst case, an attacker might even be able to leverage a single `ret` instruction as a switching mechanism, if the SP should by default directly point to the ROP chain. Detecting such a attacks with the proposed mechanism without a high number false positives is a very challenging task.

Second, while we have not observed any false positives in our experiments, it is possible that the approach will lead to false positives in practice. While there is usually no need to modify the SP of a program, it is a well-known fact that programmers seldom stick to conventions and best practices in the real world. Consequently, there certainly will be applications that modify the SP for one reason or the other and thus break the assumption the detection mechanism is based on. A possible approach to reduce the problem would be to provide the programmer or the system administrator with configuration options which, for instance, allow them to specify the size of the stack and

	Runtime [s]		*Detection*		*Switches*		
PROGRAM (10)	N	D	FP	FN	N	I	OVERHEAD
ls	0.001	0.001	0	0	0	**1,661**	**0.00**
ps	0.002	0.052	0	0	0	**1,662**	**19.78**
df	0.002	0.046	0	0	0	**1,661**	**22.10**
dig localhost	0.454	0.470	0	0	0	**1,660**	**0.04**
factor 121	0.001	0.001	0	0	0	**1,661**	**0.14**
cat /etc/passwd	0.000	0.001	0	0	0	**1,661**	**5.25**
sha1sum /etc/passwd	0.001	0.001	0	0	0	**1,661**	**0.00**
file /bin/ls	0.002	0.046	0	0	0	**1,661**	**22.11**
Sleep 1	1.002	1.003	0	0	0	**1,660**	**0.00**
id	0.028	0.035	0	0	0	**1,661**	**0.26**
persistent	1.607	1.839	0	0	0	**8,211**	**0.14**
PROGRAM (100)	N	D	FP	FN	N	I	OVERHEAD
ls	0.001	0.001	0	0	0	**165**	**0.00**
ps	0.002	0.008	0	0	0	**166**	**2.51**
df	0.002	0.007	0	0	0	**165**	**2.45**
dig localhost	0.454	0.475	0	0	0	**166**	**0.05**
factor 121	0.001	0.001	0	0	0	**165**	**0.00**
cat /etc/passwd	0.000	0.001	0	0	0	**165**	**1.07**
sha1sum /etc/passwd	0.001	0.001	0	0	0	**165**	**0.00**
file /bin/ls	0.002	0.007	0	0	0	**165**	**2.50**
Sleep 1	1.002	1.002	0	0	0	**165**	**0.00**
id	0.004	0.005	0	0	0	**165**	**0.25**
persistent	0.651	0.684	0	0	0	**7**	**0.05**
PROGRAM (1000)	N	D	FP	FN	N	I	OVERHEAD
ls	0.001	0.001	0	0	0	**15**	**0.00**
ps	0.002	0.003	0	0	0	**15**	**0.29**
df	0.002	0.002	0	0	0	**15**	**0.00**
dig localhost	0.424	0.454	0	0	0	**15**	**0.07**
factor 121	0.001	0.001	0	0	0	**15**	**0.00**
cat /etc/passwd	0.000	0.000	0	0	0	**15**	**0.00**
sha1sum /etc/passwd	0.001	0.001	0	0	0	**15**	**0.00**
file /bin/ls	0.002	0.002	0	0	0	**15**	**0.00**
Sleep 1	1.002	1.002	0	0	0	**15**	**0.00**
id	0.002	0.002	0	0	0	**15**	**0.00**
persistent	0.529	0.531	0	8	0	**1**	**0.00**

Table 7.7: The results of the three sets of experiments that we conducted with our stack switch detection mechanism. In each set we reduced the number of `ret` instructions that were trapped. While we trapped every tenth return in the first set, we only trapped every hundredth and every thousandth return in the second and third set respectively. In each set, each program was executed 100 times. The columns show from left to right: the executed program, the runtime in seconds of the normal (N) and the monitored execution (D), the false positives (FP) and false negatives (FN) that occurred, the number of stack switches seen during normal (N) and infected (I) execution, and the overhead of the monitoring.

the initial values that the SP can take. This would enable the monitoring application to distinguish between benign and malicious stack switches.

7.4.3.3 Related Work

To the best of our knowledge, the idea of detecting data-only attacks based on the value of the SP was first proposed by Fratric, who implemented a similar detection approach as we did in his system ROPGuard [46]. Fratric's ideas were later picked up by Microsoft and integrated into the Enhanced Mitigation Experience Toolkit (EMET) [92], when ROPGuard won the second prize in Microsoft's BlueHat Prize contest [94]. This shows that the proposed approach is not only effective in theory, but is also considered to be effective in practice.

There are two key differences between our approach and ROPGuard/EMET. First, ROPGuard only validates the value of the SP when a "critical“ function is called. The list of critical functions must thereby be provided as a configuration parameter to ROPGuard. This enables an attacker to evade detection by either avoiding critical functions or restoring the SP before invoking a critical function. In contrast to ROPGuard, our systems is immune to both of these attack vectors. Instead of only checking the SP when a critical function is invoked, our system can employ an arbitrary branch instruction to perform the check. In addition, an attacker cannot predict when this check is going to occur, since the hypervisor can arbitrarily modify the number of instructions that pass between two checks. For additional security, it could even randomize the number of instructions that pass between checks.

Second, ROPGuard relies on information stored by the OS to perform the actual check of the SP. In particular, ROPGuard compares the value of the SP to the value of the SP stored in the thread information block [46]. Consequently, ROPGuard can only remain functional as long as the OS is with integrity. In fact, should an attacker manage to overwrite the original SP location in the thread information block, the entire detection approach is circumvented. In contrast to ROPGuard, our approach is based on X-TIER and additionally does not rely on any information stored within the monitored system. As a result, it can remain effective even if the kernel of the monitored system is compromised.

7.4.3.4 Summary

In this section, we discussed the possibility of detecting data-only malware based on the switching mechanism it leverages for its activation. In the process, we primarily focused on detecting the behavior of a switching mechanism in order to detect known as well as unknown switching mechanisms. The conducted experiments show that the proposed mechanism is capable of reliably detecting data-only malware and only incurs a small performance overhead in practice.

7.4.4 Combining the Mechanisms

So far we considered each detection mechanism individually to be able to highlight their advantages and disadvantages. However, naturally the mechanisms could also be combined to achieve a higher level of protection. In fact, the mechanisms perfectly complement one another. To see this, let us once more consider the example of ROP. Switching-based detection hinders the attacker from placing the control structure into an arbitrary memory location and setting the SP to it. Consequently, the attacker must place the control structure on the stack to evade the mechanism. However, this makes the attack vulnerable to our second detection mechanism, which is capable of validating the layout of the stack to identify control structures. By validating the stack, we thus force the attacker to design her control structure in such a way that it would reflect a valid stack layout. But even if the attacker should manage that, this does not protect her from CRT-based detection. Therefore she must additionally only leverage gadgets during her attack that will not be detected by this approach. In summary, to evade the mechanisms an attacker must load a control structure directly onto the stack, whose layout represents a valid stack according to the application's CFG, and which only uses gadgets that do not violate the rules of the previously described shadow stack.

As one can see, the combination of all mechanisms places much more constraints on the attacker as if just a single technique would be leveraged. While it is in theory still possible for an attacker to evade all three mechanisms, doing so in a practical scenario is certainly a difficult problem. After all, the attacker is in practice not only restricted by the detection approaches, but also by the host application that is attacked and the infection mechanism (e.g. vulnerability) used.

7.5 Containment

The final defense layer within our system defense model is the containment layer. The reasoning behind this layer is that it is highly unlikely that malware infections can be prevented in general. Thus, instead of solely trying to prevent and detect malware infections, we should also try to limit the amount of damage that malware can do, once it managed to infect a system. Since malware is potentially able to infect the OS, this essentially requires us to find a way to protect applications executing within a potentially malicious system from a security layer such as the hypervisor.

In the following, we briefly describe three commonly used approaches to realize containment from the hypervisor. As containment is not specific to data-only malware, but to malware in general, however, we will only cover these approaches on an abstract level. Our intention is thereby not to provide the reader with an exhaustive overview of the field, which would be well beyond the scope of this thesis, but rather to give an impression of what is possible and to provide a starting point for further research.

7.5.1 Isolation

One of the most important security features that virtualization provides is *isolation*. Due to isolation each VM is only able to access its own memory region and files while it is unable to access the data of other VMs or the hypervisor. The idea behind *strong isolation* is to extend this approach of strictly isolated domains from the VM level to the application level. Qubes OS[8], for instance, executes each application in its own VM. As a result, each application – just like a VM – becomes unable to access the files and memory regions of other applications. Since this property is ensured by the hypervisor, this is even the case if the OS is compromised. Consequently, if such an approach is used, malware is only able to access the data of the single application it infects. The data of all other applications remains safe.

While powerful in theory, the approach has similar to other existing countermeasures limitations in practice. Most importantly, applications in general need to communicate with each other to function. For instance, may an e-mail program require access to a calender application and vice versa. To allow such functionality, we have to provide communication channels between the VMs. While we can control these communication channels from the hypervisor and enforce access controls or information flow policies, they effectively reduce the isolation between the individual VMs and thus provide an attack surface which malware can leverage to spread from one VM to another. Thus, we face a classical trade-off between security and functionality.

7.5.2 Encryption

Besides isolating applications from each other, we can also encrypt processes from the hypervisor while they are not executed by CPU. As a consequence, other processes will only be able to access the encrypted state of a process. Such an approach has, for instance, been presented by Chen et al. [26] with their system Overshadow.

In contrast to the case of strong isolation, malware will in this case be able to access the data of other processes, but since this data will be encrypted it cannot really make use of it unless it can break the encryption. This can be avoided by leveraging a secure encryption algorithm in combination with random long keys, which can be stored on the hypervisor level.

As before, encryption-based approaches face the general problem that applications often want to share data. To solve this problem, either a shared key can be leveraged that encrypts the shared data so that both applications have access to it or the data can be stored unencrypted, which is less secure though. In addition, encryption in general leads to a noticeable overhead, since the hypervisor must encrypt the current application and decrypt the next application on every context switch.

[8]http://qubes-os.org/

7.5.3 Security Modules

While strong isolation and encryption are in general enforced by the hypervisor transparent for the protected applications, an alternative approach to realize containment is to provide applications with the possibility to decide themselves which operations should be executed securely and which data should be protected. One way to achieve this is using *security modules*. For this purpose, the hypervisor enables applications to load specific modules, security modules, that will be executed in isolation from all other applications and can thus perform security critical tasks such as cryptographic operations or the storage of sensitive data. For example, could an e-mail client have a security module that encrypts and decrypts messages on its behalf. As a consequence, malware would no longer be able to extract the secret key of the e-mail application.

The main disadvantage of the approach is that each application must be specifically designed to use security modules. Thus the approach cannot be used to protect existing applications. In addition, the security module must perform all measures that are necessary to protect an application itself. For instance, in the example above the module must ensure that an attacker cannot simply leverage it as an oracle for the decryption of e-mails. To do so, it must validate whether it has been invoked by the e-mail client and if the e-mail client is with integrity and thus allowed to invoke the module.

Although the approach places the security burden on the modules, this course of action also has some advantages. First and foremost, each security module can implement a fine-granular security policy that is specialized for the application it protects. Since the developer of the security module probably has a much deeper knowledge of the application than the developer of a generic hypervisor-based containment mechanism, this will likely provide a more effective protection for each application.

Second, the overhead and the complexity of the approach is small compared to generic solutions as the hypervisor must only provide a basis for the loading, storing, and execution of specific modules. The decision of what parts of the application are protected and how is left to the application developer.

An example of an approach that provides applications with the possibility of loading and executing security modules is TrustVisor [91]. Since X-TIER already provides the possibility of executing modules securely within a guest system, it would, however, also be straightforward to implement such a mechanism on top of our framework.

7.6 Summary

In this chapter, we presented multiple approaches to counteract the threat of data-only malware. For this purpose, we first introduced a system defense model as a conceptual foundation for our countermeasures. Employing the principle of defense in depth, the model is designed to improve the resistance of current systems against malware attacks by introducing three layers of defense: prevention, detection, and containment. The key idea behind these layers is to create a network of defense mechanisms by installing countermeasures on each layer. Pursuing this defense strategy, we discussed defense mechanisms against data-only malware on each layer. In the process, we particularly focused on detection mechanisms against this novel threat as they are the key to mitigating it. To this end, we presented three concrete detection mechanisms for data-only malware: CRT-based detection, control structure-based detection, and switch-based detection. Each of these approaches relies on X-TIER for protection and employs one of the fundamental dependencies of data-only malware against it. As our experiments show, this results in effective countermeasures that each in itself can significantly hinder data-only malware. This illustrates the power of VMI and malware dependencies for malware detection. By combining the countermeasures on each layer, we can create an effective defense network against data-only malware that can mitigate the threat in many cases.

Chapter 8

Conclusion

The threat of malware has become considerably worse over the course of the last few years. One important reason for this development is that the motivation behind the creation of malware has changed drastically. While malware was formerly created as an aimless prank, its creation is now motivated by economic, political, or military objectives. As a result, the amount and the sophistication of malware has increased dramatically. In the meanwhile, we see thousands of new malware samples appearing every day that use ever more cleverly devised techniques to avoid detection.

In spite of this development, we still primarily rely on reactive mechanisms for malware defense. Instead of closing attack vectors before they can be exploited, we focus on defending against known attacks. Our method of choice thereby remains signature-based detection. As a consequence, current systems remain defenseless against novel malware forms until signatures have been created for them. However, due to the ever increasing number of malware instances, it becomes increasingly difficult for antivirus vendors to keep up with the pace of malware creation. As a result, our defenses continue to fall more and more behind. This situation has now gone so far that antivirus software is in the meantime only able to detect 51% of all new malware samples [166].

In order to counteract this development, we proposed moving away from reactive malware defense towards *proactive* malware defense. Instead of waiting for attackers to come up with ever more sophisticated malware, we feel that it is necessary to perform offensive research ourselves so that we can find weaknesses in our defenses before they are discovered by malware creators. This will enable us to identify, analyze, and mitigate future threats before they are actively exploited.

Following this approach, we analyzed current systems for weaknesses in their defenses and found that existing protection measures against malware rely in general on the assumption that malware consists of executable instructions. This, however, makes them vulnerable to data-only malware, which reuses the instructions that existed *before* its presence to provide its malicious functionality. Despite this capability and the substantial risk associated with it, data-only malware has not been researched in detail to date. As a consequence, we are currently unable to assess the overall risk that data-only malware poses let alone to counteract this potentially dangerous threat. Pursuing a proactive approach, we set out in this thesis to remedy this shortcoming by providing the first comprehensive study of data-only malware.

8.1 Contribution

Capabilities and Limitations (Q1). To be able to assess the risk that data-only malware poses, we first needed to determine the actual capabilities and limitations of this malware form. Of particular importance is hereby the question whether data-only malware poses as powerful and as realistic a threat as traditional malware does in spite of the fact that it relies solely on code reuse to function. Could data-only malware, in the course of time, even replace traditional malware in terms of its importance?

To answer these questions, we performed an in-depth analysis of data-only malware in Chapter 3. We began by studying the fundamental properties of data-only malware. In this context, we established that data-only malware can employ the same infection mechanisms as traditional malware and can infiltrate a system by either exploiting a vulnerability or infecting a file.

Having determined how data-only malware can actually penetrate a system, we moved on to discuss the individual data-only malware types that exist. Based on the execution strategies that are known from traditional malware, we distinguished between three types of data-only malware: one shot data-only malware, persistent data-only malware, and resident data-only malware. We showed that data-only malware can – similar to traditional malware – utilize all of these execution strategies. For this purpose, we outlined the challenges involved in the creation of each type and presented possibilities as to how one might overcome them. Most importantly, we showed that persistence is indeed possible in the case of data-only malware, a subject which had previously been a point of contention among researchers.

To determine whether data-only malware is practical, we followed our theoretical discussion of the individual data-only malware types by presenting detailed POC implementations. To prove the validity of our concepts, we implemented both infection approaches as well as each type of data-only malware. In the process, we took special care to provide realistic scenarios for our experiments. To this end, we infected recent Linux systems with all default protections enabled, used real world vulnerabilities to load the data-only malware, and implemented similar functionality to that which sophisticated real world rootkits provide. We found that data-only malware is as equally practical a threat as traditional malware and is able to infect current systems in spite of the various protection mechanisms that they use.

Having considered the infection and execution of data-only malware in theory and in practice, we concluded our analysis by discussing multiple advanced properties of data-only malware. Most importantly, we analyzed the computational ability and the dependencies of data-only malware in comparison to traditional malware. While data-only malware can achieve Turing completeness in many cases, we found that it has more dependencies than traditional malware. In particular its application dependency, its control structure dependency, and its dependency on a switching sequence seem well-suited as a basis for detection mechanisms.

Taking all our results into consideration, we came to the conclusion that data-only malware is indeed a realistic and powerful threat that is well-suited for the creation of sophisticated attacks.

Existing Defenses (Q2). Having determined that data-only malware is a potential risk, naturally the question emerged as to how dangerous data-only malware is for current systems considering the numerous defense mechanisms that they employ. To answer this question, we performed an analysis of the main malware defense approaches available to date in Chapter 4. In particular, we discussed current protection mechanisms such as ASLR as well as popular detection mechanisms such as signature-based detection, anomaly-based detection, integrity-based detection, and hook-based detection. While many of these concepts are in theory applicable to data-only malware, we found that most either face severe limitations in practice or have the same shortcomings as in the case of traditional malware. We concluded that hook-based detection is currently the *only* approach that could be effective against data-only malware.

However, to successfully counteract the threat of data-only malware, it is essential that a detection mechanism is not only able to detect the malware form, but also that it can do so reliably and is not prone to evasion attacks. To ensure these properties for hook-based detection, we once more employed a proactive approach and investigated the security of hook-based detection mechanisms in more detail. In the process, we found that current approaches commonly rely on the false assumption that an attacker can only modify *persistent* control data to install hooks. This makes current hook-based detection approaches vulnerable to evasion attacks in which an attacker modifies *transient* control data such as return addresses instead. To demonstrate this, we presented a novel hooking approach that we coined dynamic hooking in Chapter 5. In contrast to previous mechanisms, dynamic hooks realize the control flow transfer by triggering vulnerabilities at runtime. The hook itself will thereby reside within non-control data. Consequently, there is no evident connection between a dynamic hook and the control flow change it facilitates, which enables dynamic hooks to evade existing detection mechanisms. Moreover, due to this approach, dynamic hooks are able to perform control flow attacks as well as pure data attacks. As a result, this hooking form is not only more difficult to detect than traditional hooks, but is also more powerful than its predecessors.

Since, due to dynamic hooking, hook-based detection is also not able to defend against the threat of data-only malware, we concluded that data-only malware is not only a realistic and powerful threat, but that it also constitutes an immediate danger to current systems.

Countermeasures (Q3). This insight led to our last research question: how can we protect current systems against the threat of data-only malware?

To provide a comprehensive approach, we addressed this issue systematically. In the first step, we established a foundation that countermeasures against data-only malware can rely on. For this purpose, we introduced in Chapter 6 a framework called X-TIER, which provides a secure and flexible basis for malware detection and removal.

To provide strong security guarantees, X-TIER makes use of VMI. In particular, X-TIER enables security applications to operate from outside the system they are attempting to protect. Due to this approach, security applications run isolated from the system they are monitoring, which allows them to remain functional even if the monitored system is under the complete control of an attacker. While security applications are strongly isolated and cannot be accessed by the guest, the same does not apply in reverse. On the contrary, security applications based on X-TIER have access to the complete and untainted state of the guest system. To bridge the semantic gap, our framework provides security applications with the possibility of injecting kernel modules from the hypervisor into a running VM. Once injected, the module can then access arbitrary kernel data structures and functions while being securely executed within the untrusted guest. In the process, it can send arbitrary information back to the security application residing on the hypervisor level, which effectively enables the application to circumvent the semantic gap. This combination of security and accessibility makes X-TIER well-suited for a wide range of applications. We demonstrated this capability by presenting various example applications including a virus scanner and a malware removal tool.

Equipped with a secure and flexible foundation for our countermeasures against data-only malware, we then focused on the design of the countermeasures themselves in Chapter 7. For this purpose, we first of all formulated a system defense model. While X-TIER functions as a technical basis, this model provides the theoretical foundation for our countermeasures. In other words, it defines our overall strategy against data-only malware.

The key idea behind this model is to employ a defense in depth approach to increase the resilience of current systems against malware attacks. To accomplish this, the model defines three layers of defense: prevention, detection, and containment. By installing defense mechanisms on each of these layers, we can create a defense network against the threat of data-only malware. As a consequence, an attacker is forced to overcome a multitude of security mechanisms spread across different layers within the system in order to successfully compromise it.

Further pursuing this idea, we discussed possible countermeasures against data-only malware on each of the layers that the model proposes. In the process, we particularly focused on the detection layer and presented three concrete detection mechanisms for data-only malware: CRT-based detection, control structure-based detection, and switching mechanism-based detection. To render the evasion of these mechanisms more difficult, we based each of the countermeasures on an inherent dependency of data-only malware. Our experiments show that this approach results in valuable defense mechanisms that can, particularly when used in combination, as our model suggests, mitigate the threat of data-only malware in many cases.

8.2 Future Research Directions

Throughout the thesis, we encountered various open problems that suggest interesting future research directions. In the following, we provide a chronological overview of these problems.

Code Reuse. Code reuse techniques provide the foundation for data-only malware. However, all of the techniques presented within this thesis rely on the properties of specific instructions in order to function. This raises the question whether purely software-based code reuse approaches are possible. SROP provides the first step in this direction, but still relies on the properties of the `ret` instruction to link the execution of gadgets. Using SROP as a basis, future research could thus try to find a purely software-based code reuse technique and study its possibilities with regard to data-only malware. In this context, it would be especially interesting to determine whether software-based approaches could be used to evade CRT-based detection as we speculated.

Data-only Malware. One of the main problems of data-only malware is software diversity. Currently, data-only malware is closely bound to a specific host application. This makes it difficult to make general statements about the malware form, since the capabilities of data-only malware ultimately depend on the gadgets that are available to it. To solve this problem, we either need more large-scale studies of data-only malware, which enable us to reduce the effects of software diversity, or we must find a way to create platform independent data-only malware. To this end, we would require highly flexible code reuse approaches that are capable of creating a data-only program on the fly based on instructions that are currently available in memory. ROP compilers such as the one presented by Hund [61] could here provide a starting point for future research.

Existing Defenses. While we analyzed the effectiveness of existing countermeasures against data-only malware, we identified two approaches that would be well-suited for the detection of data-only malware, but which currently face practical limitations: CFI checking and data integrity checking. Both of them could possibly provide a good starting point for future research. The former, CFI, detects malware based on control flow changes. Since data-only malware must, similar to traditional malware, change the control flow at some point in time, the mechanism is in theory able to detect data-only malware reliably. Current implementations, however, are quite slow and overapproximate. As discussed, the latter property in particular has severe security implications that future research must address in order to provide real security based on CFI.

Data integrity checking attempts to detect malware by validating mutable data regions. Since data-only malware resides within such regions, the approach could effectively mitigate the threat. Current systems, however, are only able to validate data integrity constraints, while the constraints themselves must be generated manually. Because the approach stands and falls with its integrity constraints, future research should focus on determining novel ways to create reliable integrity constraints for data automatically. This could lead to effective countermeasures against both data-only malware and dynamic hooks.

Dynamic Hooks. We mainly explored the capabilities of dynamic hooks, but did not discussed concrete countermeasures against this hooking form, which is certainly an important future research direction. The prototype that we implemented for the discovery of dynamic hooks could here provide a starting point for additional research. In particular, instead of utilizing the prototype solely for finding dynamic hooks, we could apply it to generate a list of possible locations that are vulnerable to dynamic hooks and must thus be protected.

In this context, it would also be interesting to improve the functionality of our prototype so that is capable of finding dynamic hooks more reliably. In particular, finding a way to automatically determine the properties of binding and coverage for a hook would be a valuable contribution. In addition, one could extend the prototype to automatically generate exploits or detection rules for a particular hook.

X-TIER. Our framework provides a secure and flexible foundation for malware detection and removal. However, it can only provide this functionality for virtualized environments. In spite of the fact that most mainstream CPUs in the meanwhile support virtualization, many systems still run natively on the hardware. An interesting extension to X-TIER would thus be to provide the capability to deploy our framework on the fly to an existing system without disrupting its functionality. This would enable system administrators to inspect unvirtualized systems with X-TIER in the event that they observe suspicious behavior (e.g on the network), while users could for the most part still operate outside of a virtualized environment.

Countermeasures. While the detection mechanisms presented within this thesis will significantly raise the bar for an attacker, it is unlikely that they can stop data-only malware in general. Thus it is indispensable that future research improves the proposed mechanisms or proposes additional mechanisms against data-only malware.

In this thesis, we primarily focused on the detection of data-only malware, while we only covered prevention and containment from a theoretical point of view. To increase the resistance of systems against attacks, we require novel defenses on these layers as well. In the case of malware prevention, we especially consider architectural prevention mechanisms as an interesting field for future research. The prevention mechanisms that we suggested in Section 7.3.2 could here provide an initial starting point for the development of novel countermeasures.

However, even more important in our opinion is research in the area of containment. Instead of focusing on the attack itself, containment attempts to address the issue of what can be done when an attack succeeds. Since, due to the complexity of modern systems, it is becoming more and more difficult to close all attack vectors, we consider the handling of incidents as a crucial aspect in future defense.

8.3 Final Words

In this thesis, we set out to proactively analyze and mitigate a dangerous future threat: data-only malware. This form of malware is unique in that it can infect a system without introducing a single instruction. To accomplish this, data-only malware combines existing instructions to create a new program. As our analysis has shown, this approach enables this malware form to evade many proven and tested defense mechanisms. In addition, we found that data-only malware can in general perform Turing complete computations. This empowers the malware form to infect current systems and to execute sophisticated attacks. We illustrated this by implementing and evaluating several instances of data-only malware in real world scenarios. We then presented a general VMI-based framework for the detection and removal of malware to counteract the threat. Employing this framework and the insights that we obtained in the course of our analysis, we suggested novel defense mechanisms that detect data-only malware by exploiting its fundamental dependencies. This approach resulted in strong initial countermeasures that can mitigate the threat in many cases.

Our work leads to four key insights: first, offensive research is indispensable in order to increase the security of current systems. By assuming the role of the attacker and questioning the reliability of existing approaches, we can identify and eliminate weaknesses before they can be exploited. This will allow us to gradually improve our defenses and increase the overall security of our systems. Throughout the thesis we illustrated multiple weaknesses in current defenses and proposed mechanisms to improve them.

Second, we showed that the common approach to system defense is fundamentally flawed. The main reason for this is that this approach rests on the assumption that existing defenses can prevent malware infections. However, as data-only malware amply demonstrates, existing defense mechanisms cannot provide this capability. To solve this problem, we must improve existing defense mechanisms and fundamentally rethink our approach to malware defense. A key technology in this regard is certainly VMI, which enables security applications to remain functional even when the entire system they monitor is compromised. We demonstrated this capability in our X-TIER framework. Based on this framework, we suggested a new model for malware defense that employs multiple security layers that make compromising a system significantly harder.

Third, dependencies are a crucial weakness of malware that is predestined for its detection. For example, if we hinder data-only malware from obtaining the addresses of the instructions it requires, the malware will no longer be able to infect the system. Employing this insight, we identified several dependencies of data-only malware that allowed us to create effective countermeasures.

Fourth, the assumptions that an approach makes are decisive for its security. As soon as a mechanism relies on incorrect assumptions, it is likely to be circumvented. We illustrated this by effectively evading hook-based detection mechanisms with dynamic hooks.

In conclusion, we showed that data-only malware is a realistic and dangerous threat which could replace traditional malware in the course of time. By providing a comprehensive study about data-only malware, we laid the foundation for the defense against this malware form.

Acronyms

A

ABI	application binary interface	22, 27, 34, 39–41, 160, 161
API	application programming interface	40, 55
APIC	Advanced Programmable Interrupt Controller	181, 182
ASLR	address space layout randomization	50, 56, 66, 84–86, 100, 110, 111, 117, 121, 172, 174, 176, 215

B

BTF	single-step on branches flag	180
BTS	branch trace store	184

C

CCC	Chaos Computer Club	10
CERT	computer emergency response team	11
CFG	control flow graph	57, 58, 114, 115, 118, 126, 195–197, 209
CFI	control flow integrity validation	28, 57, 114, 115, 118, 192, 217
CFL	control flow locking	114, 115
CPL	current privilege level	51, 150
CPU	central processing unit	21, 36, 54, 55, 111, 163, 176, 185, 187, 196, 198, 199, 203, 205, 210, 218

D

DARPA	U.S. Defense Advanced Research Projects Agency	11
DDOS	distributed denial of service	1, 14
DKOM	direct kernel object manipulation	58, 140, 166
DOS	disk operating system	10
DR	debug register	179, 180

E

ELF	Executable and Linkable Format 16, 23, 66, 84–86, 88, 90, 91, 97, 100, 107, 146, 161, 175, 197
EMET	Enhanced Mitigation Experience Toolkit 208
EPT	Extended Page Tables 152, 157, 179

F

FP	frame pointer 28, 36, 82, 90, 92, 95, 96, 150

G

GDT	global descriptor table 75
GOT	global offset table 78, 88, 89
GS	global segment register 127

I

IAT	Import Address Table 78
IDS	intrusion detection system 42
IDT	interrupt descriptor table 43, 75, 76
IDTR	interrupt descriptor table register 151
IF	interrupt enable flag 151, 156
ILM	instruction-level monitoring 178–180, 182–184, 187, 190–192
IP	instruction pointer 27, 28, 34, 36, 37, 62–64, 67, 72–76, 90, 92, 96, 102, 105, 150, 152, 160, 173, 175, 183, 184, 186, 187, 203
IPS	intrusion prevention system 42
IRC	internet relay chat 14
ISA	instruction set architecture 38–41, 43
IST	Interrupt Stack Table 75, 76

J

JIT	just-in-time 175
JOP	jump-oriented programming 30–32, 34–36, 106, 190, 206
JVM	Java virtual machine 38, 40

K

KASLR	kernel ASLR 50

L

LBR	last branch record stack 183, 184

M

MS-DOS	Microsoft disk operating system	10
MSR	model-specific register	74, 92, 110, 117, 181, 183, 184
MTF	monitor trap flag	180

N

NMI	non-maskable interrupt	182
NOP	no operation	34, 77

O

OS	operating system	3, 7, 14, 18, 20, 22, 32, 37, 39–41, 43–50, 57, 63, 74, 75, 77, 98, 99, 101, 102, 107, 110, 114, 116, 126, 128, 135, 143–149, 151, 154, 155, 159–167, 170–172, 180, 185, 187, 197, 199, 205, 208–210

P

PC	personal computer	10
PE	Portable Executable	23, 85, 146
PES	performance event select	181
PF	page fault	179
PMC	performance monitoring counter	179–184, 191
PMU	performance monitoring unit	180
POC	proof of concept	5, 7, 61, 62, 82, 84, 86, 91, 93, 95, 97, 120, 128, 130, 136, 139, 214

R

ROP	return-oriented programming	3, 28–30, 32, 35, 36, 50, 51, 63, 64, 66, 67, 71–74, 76–79, 82, 91, 92, 102, 104–106, 115, 121, 177, 178, 186, 189–192, 203, 204, 206, 209, 217

S

SILM	selective instruction-level monitoring	179, 184, 185, 187, 188, 191, 198, 204, 205
SMAP	supervisior mode access prevention	51, 111
SMEP	supervisior mode execution prevention	3, 51, 111
SP	stack pointer	28–31, 34, 36, 63, 66, 67, 71–78, 82, 88, 90, 92, 95, 102, 104, 105, 150, 151, 196, 197, 203–206, 208, 209
SROP	sigreturn-oriented programming	32, 35, 36, 190, 217
SSA	single static assignment form	127, 128
SSDT	System Service Dispatch Table	110

T

TCB	trusted computing base	37
TF	trap flag	179, 180, 187
TOS	top-of-stack pointer	183, 184
TSS	task-state segment	75

V

VM	virtual machine	6, 20, 38, 40–44, 47, 49, 84, 141–145, 147–153, 155, 156, 158, 159, 163–165, 167, 181–184, 188, 210, 216
VMI	virtual machine introspection	7, 9, 37, 38, 40–44, 46, 49, 59, 138, 140, 141, 143, 156, 158, 164, 167, 212, 216, 219
VMM	virtual machine monitor	41, 43, 48

X

XD	execute-disable	194

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow Integrity". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-Flow Integrity Principles, Implementations, and Applications". In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (Oct. 2009), pp. 1–40.

[3] Aleph One. "Smashing The Stack For Fun And Profit". In: *Phrack* 49.14 (Aug. 1996).

[4] James P. Anderson. *Computer Security Technology Planning Study.* Tech. rep. Hanscom Air Force Base, 1972.

[5] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. "DKSM: Subverting Virtual Machine Introspection for Fun and Profit". In: *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems.* IEEE, 2010, pp. 82–91.

[6] M. Bailey, E. Cooke, F. Jahanian, D. Watson, and J. Nazario. "The Blaster Worm: Then and Now". In: *IEEE Security and Privacy Magazine* 3.4 (July 2005), pp. 26–31.

[7] A. Baliga, V. Ganapathy, and L. Iftode. "Detecting Kernel-Level Rootkits Using Data Structure Invariants". In: *IEEE Transactions on Dependable and Secure Computing* 8.5 (2011).

[8] Arati Baliga, Pandurang Kamat, and Liviu Iftode. "Lurking in the Shadows: Identifying Systemic Threats to Kernel Data". In: *Proceedings of the IEEE Symposium on Security and Privacy.* 2007.

[9] Boldizsar Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. "The Cousins of Stuxnet: Duqu, Flame, and Gauss". In: *Future Internet* 4 (2012).

[10] Matt Bishop. *Computer Security: Art and Science.* Addison-Wesley, 2002.

[11] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. "Hacking Blind". In: *Proceedings of the Symposium on Security and Privacy.* 2014.

[12] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. "Jump-Oriented Programming: A New Class of Code-Reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ACM Press, 2011.

[13] Reverend Bill Blunden. *The Rootkit ARSENAL: Escape and Evasion in the Dark Corners of the System.* Wordware Publishing, 2009.

[14] Jean-Marie Borello and Ludovic Mé. "Code obfuscation techniques for metamorphic viruses". In: *Journal in Computer Virology* 4.3 (Aug. 2008), pp. 211–220.

[15] Erik Bosman and Herbert Bos. "Framing Signals–A Return to Portable Shellcode". In: *Proceedings of the IEEE Symposium on Security and Privacy.* 2014.

[16] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 2006.

[17] Ralf Burger. *Computer Viruses: A High-tech Disease.* 2nd ed. Data Becker, Dec. 1988.

[18] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. "Secure and Robust Monitoring of Virtual Machines through Guest-Assisted Introspection". In: *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID).* Springer, 2012, pp. 22–41.

[19] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. "Mapping kernel objects to enable systematic integrity checking". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* ACM Press, 2009.

[20] Nicholas Carlini and David Wagner. "ROP is Still Dangerous: Breaking Modern Defenses". In: *Proceedings of the USENIX Security Symposium.* 2014.

[21] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. "Can DREs provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage". In: *Proceedings of the conference on Electronic voting technology/workshop on trustworthy elections.* USENIX Association, 2009.

[22] Peter M. Chen and Brian D. Noble. "When Virtual Is Better Than Real". In: *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems.* IEEE Computer Society, 2001.

[23] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. "DROP: Detecting Return-Oriented Programming Malicious Code". In: *Lecture Notes in Computer Science* (2009), pp. 163–177.

[24] Ping Chen, Xiao Xing, Bing Mao, and Li Xie. "Return-Oriented Rootkit without Returns (on the x86)". In: *Information and Communications Security*. Vol. 6476. LNCS. Springer, 2010, pp. 340–354.

[25] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. "Non-control-data Attacks Are Realistic Threats". In: *Proceedings of the USENIX Security Symposium*. 2005.

[26] 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". In: *Proceedings of the Conference on Architectural Support for Programming Languages and Sperating Systems (ASPLOS)* (2008).

[27] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. "ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks". In: *Proceedings of the Network and Distributed System Security (NDSS) Symposium*. Internet Society, 2014.

[28] Mihai Christodorescu, Somesh Jha, Sanjit .A. Seshia, Dawn Song, and Randal E. Bryant. "Semantics-Aware Malware Detection". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2005.

[29] Fred Cohen. "Computer Viruses - Theory and Experiments". In: *Computers & Security* 6.1 (Feb. 1987), pp. 22–35.

[30] Evan Cooke, Frarnam Jahanian, and Danny McPherson. "The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets". In: *Proceedings of the Workshop of Steps to Reducing Unwanted Traffic on the Internet*. USENIX Association, June 2005.

[31] corelanc0d3r. *Exploit writing tutorial part 11 : Heap Spraying Demystified*. Accessed: 17.04.2014. Dec. 2011. URL: `https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/`.

[32] Symantec Corporation. *Internet Security Threat Report 2014*. Tech. rep. Symantec Corporation, 2014.

[33] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks". In: *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, 1998.

[34] John Criswell, Nathan Dautenhahn, and Vikram Adve. "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels". In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2014.

[35] *CVE Details: The ultimate security vulnerability datasource*. Accessed: 27.07.2014. MITRE Corporation. URL: `http://www.cvedetails.com/`.

[36] Dino A. Dai Zovi. *Practical Return-Oriented Programming.* Accessed: 15.04.2014. Apr. 2010. URL: `http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf`.

[37] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *Proceedings of the USENIX Security Symposium.* 2014.

[38] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. "Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks". In: *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC).* ACM Press, 2009.

[39] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. "ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS).* ASIACCS '11. Hong Kong, China: ACM, 2011, pp. 40–51.

[40] 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.* ACM Press, 2008, pp. 51–62.

[41] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection". In: *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE, May 2011.

[42] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. "Robust signatures for kernel data structures". In: *Proceedings of the ACM conference on Computer and Communications Security (CCS).* ACM Press, 2009.

[43] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle.* 8th ed. Oldenbourg Verlag, 2013.

[44] Eldad Eilam. *Reversing: Secrets of Reverse Engineering.* Wiley, 2005.

[45] Peter Ferrie. *Attacks on More Virtual Machine Emulators.* Tech. rep. Symantec Advanced Threat Research, 2007.

[46] Ivan Fratric. *Runtime Prevention of Return-Oriented Programming Attacks.* Tech. rep. 2012.

[47] F-Secure. *Virus:Boot/Brain.* Accessed: 13.03.2014. URL: `http://www.f-secure.com/v-descs/brain.shtml`.

[48] Yangchun Fu and Zhiqiang Lin. "EXTERIOR: Using a Dual-VM Based External Shell for Guest-OS Introspection, Configuration, and Recovery". In: *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*. ACM Press, 2013.

[49] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. "Compatibility is Not Transparency: VMM Detection Myths and Realities". In: *Proceedings of the 11th USENIX workshop on Hot Topics in Operating Systems*. USENIX Association, 2007.

[50] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". In: *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. 2003.

[51] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization". In: *Proceedings of the 21st USENIX Security Symposium*. USENIX Association, 2012.

[52] Heather Goudey. *Microsoft Malware Protection Center, Threat Report: Rootkits*. Tech. rep. Microsoft Corporation, June 2012. URL: `http://www.microsoft.com/en-us/download/confirmation.aspx?id=34797`.

[53] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. "Automatic Generation of String Signatures for Malware Detection". In: *Recent Advances in Intrusion Detection*. Ed. by Engin Kirda, Somesh Jha, and Davide Balzarotti. Vol. 5758. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 101–120.

[54] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. "Process Implanting: A New Active Introspection Framework for Virtualization". In: *Proceedings of the 30th International Symposium on Reliable Distributed Systems*. IEEE, Oct. 2011.

[55] Mordechai Guri, Gabi Kedma, Assaf Kachlon, and Yuval Elovici. "AirHopper: Bridging the Air-Gap between Isolated Networks and Mobile Phones using Radio Frequencies". In: *Proceedings of the 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*. IEEE, Oct. 2014.

[56] Hex-Rays. *IDA Pro*. June 2014. URL: `https://www.hex-rays.com/products/ida/`.

[57] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. "Ensuring operating system kernel integrity with OSck". In: *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2011.

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

[59] Thorsten Holz. “Tracking and Mitigation of Malicious Remote Control Networks”. PhD thesis. Universität Mannheim, 2009.

[60] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. “Microgadgets: Size Does Matter In Turing-complete Return-oriented Programming”. In: *In Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*. 2012.

[61] Ralf Hund. “Countering Lifetime Kernel Code Integrity Protections”. Diploma thesis. Universität Mannheim, May 2009.

[62] Ralf Hund, Thorsten Holz, and Felix C. Freiling. “Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms”. In: *Proceedings of the USENIX Security Symposium*. 2009.

[63] Nwokedi Idika and Aditya P. Mathur. *A Survey of Malware Detection Techniques*. Tech. rep. Purdue University, 2007.

[64] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*. Intel Corporation. Feb. 2014.

[65] Izik. *Abusing .CTORS and .DTORS For FUN and PROFIT*. Tech. rep. 2006. URL: `http://www.exploit-db.com/papers/13234/`.

[66] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. “Stealthy malware detection and monitoring through VMM-based “out-of-the-box” semantic view reconstruction”. In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. Feb. 2007.

[67] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “VMM-based hidden process detection and identification using Lycosid”. In: *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. ACM Press, 2008.

[68] Jakub Kaminski and Hamish O’Dea. *How to smell a RAT - remote administration tools vs backdoor Trojans*. Accessed: 28.03.2014. 2002. URL: `http://www.virusbtn.com/conference/vb2002/abstracts/remote_administration.xml`.

[69] Martin Karresand. “A Proposed Taxonomy of Software Weapons”. MA thesis. Linköping University, 2002.

[70] Eugene Kaspersky. *MALWARE: Von Viren, Würmern, Hackern und Trojanern und wie man sich vor ihnen schützt*. Carl Hanser Verlag, 2008.

[71] Sanmeet Kaur and Maninder Singh. “Automatic attack signature generation systems: A review”. In: *IEEE Security & Privacy* 11.6 (2013), pp. 54–61.

[72] *Kernel Patch Protection: Frequently Asked Questions*. Accessed: 14.04.2014. Microsoft. Jan. 2007. URL: `http://msdn.microsoft.com/en-us/windows/hardware/gg487353.aspx`.

[73] Gene H. Kim and Eugene H. Spafford. “The Design and Implementation of Tripwire: A File System Integrity Checker”. In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* ACM Press, 1994.

[74] James C. King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM (CACM).* 1976.

[75] Evgenios Konstantinou and Stephen Wolthusen. *Metamorphic Virus: Analysis and Detection.* Tech. rep. Royal Holloway University of London, 2008.

[76] Tim Kornau. “Return Oriented Programming for the ARM Architecture”. Diploma thesis. Ruhr-Universität Bochum, Dec. 2009.

[77] Jürgen Kraus. “Selbstreproduktion bei Programmen”. Diploma thesis. Universität Dortmund, 1980.

[78] Kruegel Kruegel, William Robertson, and Giovanni Vigna. “Detecting Kernel-Level Rootkits Through Binary Analysis”. In: *Proceedings of the 20th Annual Computer Security Applications Conference.* IEEE, 2004.

[79] Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. “KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object”. In: *Proceedings of the USENIX Security Symposium.* USENIX Association, 2013.

[80] Wenke Lee and Salvatore J. Stolfo. “Data Mining Approaches for Intrusion Detection”. In: *Proceedings of the USENIX Security Symposium.* 1998.

[81] Jinku Li, Zhi Wang, Tyler Bletsch, Deepa Srinivasan, Michael Grace, and Xuxian Jiang. “Comprehensive and Efficient Protection of Kernel Control Data”. In: *IEEE Transactions on Information Forensics and Security* 6.4 (2011).

[82] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. “Defeating return-oriented rootkits with ”Return-Less” kernels”. In: *Proceedings of the 5th European conference on Computer Ssystems.* ACM, 2010.

[83] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and B. Chavez. “Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE, 2006.

[84] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. “SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures”. In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS).* USENIX Association, 2011.

[85] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 7 Edition.* Oracle. Feb. 2013. URL: `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`.

[86] Felix Lindner. *Router Exploitation.* Accessed: 15.04.2014. Rsecurity Labs. Nov. 2009. URL: `http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf`.

[87] Lionel Litty, H. Andres Lagar-Cavilla, and David Lie. “Hypervisor Support for Identifying Covertly Executing Binaries”. In: *Proceedings of the 17th USENIX Security Symposium.* USENIX Association, 2008.

[88] Corey Malone, Mohamed Zahran, and Ramesh Karri. “Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs”. In: *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC).* 2011.

[89] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface.* Tech. rep. 2013.

[90] Wolfgang Mauerer. *Das ELF-Binärformat.* Tech. rep. 2003. URL: `http://www.linux-kernel.de/appendix/ap05.pdf`.

[91] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. “TrustVisor: Efficient TCB Reduction and Attestation”. In: *Proceedings of the IEEE Symposium on Security & Privacy.* IEEE, 2010.

[92] Microsoft. *Enhanced Mitigation Experience Toolkit (EMET).* Accessed: 27.07.2015. URL: `https://support.microsoft.com/de-de/kb/2458544`.

[93] Microsoft. *Microsoft PE and COFF Specification.* Accessed: 01.04.2014. Feb. 2013. URL: `http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx`.

[94] Microsoft. *Microsoft Security Toolkit Delivers New BlueHat Prize Defensive Technology.* Accessed: 27.07.2015. July 2012. URL: `http://news.microsoft.com/2012/07/25/microsoft-security-toolkit-delivers-new-bluehat-prize-defensive-technology/`.

[95] Microsoft-Research. *Z3: Theorem Prover.* June 2014. URL: `http://z3.codeplex.com/`.

[96] David Moore, Colleen Shannon, and Jefferey Brown. “Code-Red: a case study on the spread and victims of an Internet worm”. In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment.* ACM Press, 2002.

[97] Andreas Moser, Christopher Kruegel, and Engin Kirda. “Limits of Static Analysis for Malware Detection”. In: *Proceedings of the 23rd Annual Computer Security Applications Conference.* IEEE, 2007.

[98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. “An Empirical Study of Static Call Graph Extractors”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7.2 (Apr. 1998), pp. 158–191.

[99] Carey Nachenberg. “Computer Virus-Antivirus Coevolution”. In: *Communications of the ACM* 40.1 (Jan. 1997), pp. 46–51.

[100] Nergal. "The advanced return-into-lib(c) exploits: PaX case study". In: *Phrack* 58.4 (Dec. 2001).

[101] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2007.

[102] Anh M Nguyen, Nabil Schear, Heedong Jung, Apeksha Godiyal, Samuel T King, and Hai D. Nguyen. "MAVMM: Lightweight and Purpose Built VMM for Malware Analysis". In: *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. 2009.

[103] Shaun Nichols. *Kernel-level malware on the rise*. Accessed: 02.04.2014. itnews. Feb. 2007. URL: `http://www.itnews.com.au/News/73966,kernel-level-malware-on-the-rise.aspx`.

[104] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. "G-Free : Defeating Return-Oriented Programming Through Gadget-Less Binaries". In: *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. A, Dec. 2010.

[105] Tavis Ormandy. *An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments*. Tech. rep. Google, 2007.

[106] Pandalabs. *Quarterly Report January-March 2014*. Tech. rep. Pandalabs, 2014.

[107] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing". In: *Proceedings of the USENIX Security Symposium*. USENIX Association, 2013.

[108] Jeremy Paquette. *A History of Viruses*. Accessed: 13.03.2014. Symantec. Nov. 2010. URL: `http://www.symantec.com/connect/articles/history-viruses`.

[109] Corina S. Pasareanu and Willem Visser. "A Survey of New Trends in Symbolic Execution for Software Testing and Analysis". In: *International Journal of Software Tools Technology Transfer* 11.4 (2009).

[110] Animesh Patcha and Jung-Min Park. "An overview of anomaly detection techniques: Existing solutions and latest technological trends". In: *Computer Networks* 51.12 (Aug. 2007), pp. 3448–3470.

[111] PaX Team. *Supervisor Mode Access Prevention*. Accessed: 14.04.2014. Sept. 2012. URL: `http://forums.grsecurity.net/viewtopic.php?f=7&t=3046`.

[112] 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*. IEEE, May 2008, pp. 233–247.

[113] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core*. Elsevier, 2011.

[114] Nick L. Jr. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. “Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor”. In: *Proceedings of the 13th USENIX Security Symposium.* USENIX Association, 2004, pp. 179–194.

[115] Nick L. Petroni Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. “An Architecture for Specification-based Detection of Semantic Integrity Violations in Kernel Dynamic Data”. In: *Proceedings of the 15th USENIX Security Symposium.* USENIX Association, 2006.

[116] Nick L. Petroni Jr. and Michael Hicks. “Automated Detection of Persistent Kernel Control-Flowlow Attacks”. In: *Proceedings of the ACM conference on Computer and Communications Security (CCS).* ACM, 2007.

[117] Jonas Pfoh. “Leveraging Derivative Virtual Machine Introspection Methods for Security Applications”. PhD thesis. Technische Universität München, 2012.

[118] Jonas Pfoh, Christian Schneider, and Claudia Eckert. “A Formal Model for Virtual Machine Introspection”. In: *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec ’09).* ACM Press, Nov. 2009, pp. 1–10.

[119] Jonas Pfoh, Christian Schneider, and Claudia Eckert. “Exploiting the x86 Architecture to Derive Virtual Machine State Information”. In: *Proceedings of the Fourth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2010).* Best Paper Award. Venice, Italy: IEEE Computer Society, July 2010, pp. 166–175.

[120] Jonas Pfoh, Christian Schneider, and Claudia Eckert. “Nitro: Hardware-based System Call Tracing for Virtual Machines”. In: *Advances in Information and Computer Security.* Vol. 7038. Lecture Notes in Computer Science. Springer, 2011, pp. 96–112.

[121] Check Point. *Check Point Security Report 2014.* Tech. rep. Check Point, 2014.

[122] Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. “Manipulating semantic values in kernel data structures: Attack assessments and implications”. In: *Proceedings of the Conference on Dependable Systems and Networks (DSN).* June 2013.

[123] Oxford University Press. *malware: definition of malware in the Oxford Dictionary (British & World English).* Oxford University Press. Jan. 2015. URL: `http://www.oxforddictionaries.com/definition/english/malware`.

[124] Oxford University Press. *Programme: Definition of Programme in Oxford Dictionary (British & World English).* Oxford University Press. July 2014. URL: `http://www.oxforddictionaries.com/definition/english/programme`.

[125] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. “Camouflage in Malware: from Encryption to Metamorphism”. In: *International Journal of Computer Science and Network Security* 12.8 (Aug. 2012), pp. 74–83.

[126] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. “Security in Embedded Systems”. In: *ACM Transactions on Embedded Computing Systems* 3.3 (2004).

[127] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. “Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring”. In: *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2009, pp. 74–81.

[128] Ryan Riley, Xuxian Jiang, and Dongyan Xu. “Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing”. In: *Lecture Notes in Computer Science*. Vol. 5230. Springer, 2008, pp. 1–20.

[129] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. “Return-Oriented Programming: Systems, Languages, and Applications”. In: *ACM Transactions on Information and System Security (TISSEC) - Special Issue on Computer and Communications Security* 15.1 (Mar. 2012), pp. 1–34.

[130] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals - Part I*. 6th. Microsoft Press, 2012.

[131] Joanna Rutkowska. *Introducing Stealth Malware Taxonomy*. Tech. rep. COSEINC Advanced Malware Labs, Nov. 2006.

[132] Tim Schiesser. *Symantec claims anti-virus is ”dead”*. Accessed: 26.06.2014. TECHSPOT. 2014. URL: `http://www.techspot.com/news/56656-symantec-claims-anti-virus-is-dead.html`.

[133] Christian Schneider. “Full Virtual Machine State Reconstruction for Security Applications”. PhD thesis. Technische Universität München, 2013.

[134] Christian Schneider, Jonas Pfoh, and Claudia Eckert. “Bridging the Semantic Gap Through Static Code Analysis”. In: *Proceedings of the 5th European Workshop on System Security (EuroSec’12)*. ACM Press, 2012.

[135] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. “Evaluating the Effectiveness of Current Anti-ROP Defenses”. In: *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2014.

[136] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2010.

[137] Asaf Shabtai, Eitan Menahem, and Yuval Elovici. “F-Sign: Automatic, Function-Based Signature Generation for Malware”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 41.4 (2011), pp. 494–508.

[138] Hovav Shacham. “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”. In: *Proceedings of the 14th conference on Computer and Communications Security*. ACM, 2007.

[139] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. “On the Effectiveness of Address-Space Randomization”. In: *Proceedings of the ACM conference on Computer and Communications Security (CCS)*. ACM Press, 2004.

[140] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. ““Weird Machines” in ELF: A Spotlight on the Underappreciated Metadata”. In: *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*. 2013.

[141] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. “Secure in-VM monitoring using hardware virtualization”. In: *Proceedings of the 16th ACM conference on Computer and Communications Security*. ACM Press, 2009.

[142] Yan Shoshitaishvili. *Python bindings for Valgrind’s VEX IR*. June 2014. URL: `https://github.com/zardus/pyvex`.

[143] sickness. *Linux exploit writing tutorial part 2: Atack Overflow ASLR bypass using ret2reg*. Accessed: 15.04.2014. Mar. 2011. URL: `http://dl.packetstormsecurity.net/papers/attack/lewt2-aslrbypass.pdf`.

[144] MIchael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. no starch press, 2012.

[145] skape and Skywing. *Bypassing PatchGuard on Windows x64*. Tech. rep. 2005.

[146] Skywing. *PatchGuard Reloaded: A Brief Analysis of PatchGuard Version 3*. Tech. rep. 2007.

[147] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, 2005.

[148] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. “Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2013.

[149] Solar Designer. *Getting around non-executable stack (and fix)*. Bugtraq Mailing List. Aug. 1997.

[150] Sophos. *Sophos Data Protection for Business*. Accessed: 13.03.2014. URL: `http://www.sophos.com/en-us.aspx`.

[151] Eugene H. Spafford. “The Internet Worm: Crisis and Aftermath”. In: *Communications of the ACM* 32.6 (June 1989), pp. 678–687.

[152] Sherri Sparks and Jamie Butler. “Shadow Walker: Raising The Bar For Windows Rootkit Detection”. In: *Phrack* 11.63 (2005).

[153] B. Sprunt. “The Basics of Performance-Monitoring Hardware”. In: *IEEE Micro* 22.4 (July 2002), pp. 64–71.

[154] Stuart Staniford, Vern Paxson, and Nicholas Weaver. “How to Own the Internet in Your Spare Time”. In: *Proceedings of the 11th USENIX Security Symposium.* USENIX Association, 2002, pp. 149–167.

[155] *System V Application Binary Interface - DRAFT.* Accessed:01.04.2014. XINUOS. June 2013. URL: `http://www.sco.com/developers/gabi/latest/contents.html`.

[156] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. “SoK: Eternal War in Memory”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* 2013.

[157] Peter Szor. *The Art of Computer Virus Research and Defense.* Ed. by Karen Gettman. Addison-Wesley, 2005.

[158] Andrew S. Tanenbaum. *Modern Operating Systems.* 3rd revised edition. Pearson Education, 2009.

[159] Neeraj Thakar. *Botnets Remain a Leading Threat.* Accessed: 31.03.2014. McAfee. Mar. 2013. URL: `https://blogs.mcafee.com/business/security-connected/tackling-the-botnet-threat`.

[160] Ken Thompson. “Reflections on Trusting Trust”. In: *Communications of the ACM* 27.8 (Aug. 1984), pp. 761–763.

[161] Frank Tip. *A Survey of Program Slicing Techniques.* Tech. rep. Amsterdam, The Netherlands, The Netherlands: Centre for Mathematics and Computer Science (CWI), 1994.

[162] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. “On the Expressiveness of Return-into-libc Attacks”. In: *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection.* Springer, 2011, pp. 121–141.

[163] *UEFI Specification.* UEFI Inc. July 2013. URL: `http://www.uefi.org/specs/download/UEFI_2_4.pdf`.

[164] Amit Vasudevan and Ramesh Yerraballi. “Stealth Breakpoints”. In: *Proceedings of the Annual Computer Security Applications Conference (ACSAC).* 2005.

[165] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, 2002.

[166] Giovanni Vigna. *AV Isn't Dead, It Just Can't Keep Up.* Lastline Labs. May 2014. URL: `http://labs.lastline.com/lastline-labs-av-isnt-dead-it-just-cant-keep-up`.

[167] Sebastian Vogl. “A bottom-up Approach to VMI-based Kernel-level Rootkit Detection”. Diploma thesis. Technische Unversität München, Oct. 2010. URL: `http://www.sec.in.tum.de/assets/studentwork/finished/Vogl2010.pdf`.

[168] Sebastian Vogl and Claudia Eckert. “Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture”. In: *Proceedings of EuroSec’12, 5th European Workshop on System Security,* ACM Press, 2012.

[169] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. “X-TIER: Kernel Module Injection”. In: *Proceedings of the 7th International Conference on Network and System Security.* Vol. 7873. Lecture Notes in Computer Science. Springer, June 2013, pp. 192–206.

[170] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. “Persistent Data-only Malware: Function Hooks without Code”. In: *Proceedings of the 21th Annual Network & Distributed System Security Symposium (NDSS).* 2014.

[171] John von Neumann. *Theory of Self-Reproducing Automata.* Ed. by Arthur W. Burks. Champaign, IL, USA: University of Illinois Press, 1966.

[172] Jiang Wang, Angelos Stavrou, and Anup Ghosh. “HyperCheck: A Hardware-Assisted Integrity Monitor”. In: *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID).* Springer, 2010.

[173] Zhi Wang and Xuxian Jiang. “HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE, 2010, pp. 380–395.

[174] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. “Countering Kernel Rootkits with Lightweight Hook Protection”. In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* ACM Press, 2009.

[175] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. “Countering Persistent Kernel Rootkits through Systematic Hook Discovery”. In: *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID).* Springer, 2008.

[176] Mark Weiser. “Program Slicing”. In: *Proceedings of the International Conference on Software Engineering (ICSE).* 1981.

[177] Andrew White. “Identifying the Unknown in User Space Memory”. PhD thesis. Queensland University of Technology - Institute for Future Environments, 2013.

[178] Georg Wicherski. *Using Performance Counters to Detect Kernel Return-Oriented-Programming.* Tech. rep. Crowdstrike, 2013.

[179] David Williams-King. “Binary Shuffling: Defeating Memory Disclosure Attacks through Re-Randomization”. MA thesis. The University of British Columbia, July 2014.

[180] Wing Wong and Mark Stamp. “Hunting for metamorphic engines”. In: *Journal in Computer Virology* 2.3 (Dec. 2006), pp. 211–229.

[181] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. “CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters”. In: *Conference on Dependable Systems and Networks (DSN).* 2012.

[182] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. “Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-Scale Object-Oriented Applications”. In: *Proceedings of the FSE/SDP workshop on Future of Software Engineering Research (FoSER).* ACM Press, 2010. URL: `http://dx.doi.org/10.1145/1882362.1882448`.

[183] Min Xu, Xuxian Jiang, Ravi Sandhu, and Xinwen Zhang. “Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection”. In: *Proceedings of the 12th ACM symposium on Access Control Models and Technologies.* ACM Press, 2007.

[184] Heng Yin, Zhenkai Liang, and Dawn Song. “Hookfinder: Identifying and understanding malware hooking behaviors”. In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS).* USENIX Association, 2008.

[185] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. “Practical Control Flow Integrity and Randomization for Binary Executables”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE, 2013.

[186] Mingwei Zhang and R. Sekar. “Control Flow Integrity for COTS Binaries”. In: *Proceedings of the USENIX Security Symposium.* 2013.