JYU DISSERTATIONS 132

Roee Shimon Leon

Applications of Hypervisors in Security

UNIVERSITY OF JYVÄSKYLÄ
FACULTY OF INFORMATION TECHNOLOGY

JYU DISSERTATIONS 132

Roee Shimon Leon

Applications of Hypervisors in Security

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi yliopiston vanhassa juhlasalissa S212
lokakuun 24. päivänä 2019 kello 10.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
in building Seminarium, auditorium S212, on October 24, 2019 at 10 o'clock.

JYVÄSKYLÄN YLIOPISTO
UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2019

Editors
Timo Männikkö
Faculty of Information Technology, University of Jyväskylä
Ville Korkiakangas
Open Science Centre, University of Jyväskylä

Permanent link to this publication: http://urn.fi/URN:ISBN:978-951-39-7854-9

ISBN 978-951-39-7854-9 (PDF)
URN:ISBN:978-951-39-7854-9
ISSN 2489-9003

ABSTRACT

Leon, Roee Shimon
Applications of Hypervisors in Security
Jyväskylä: University of Jyväskylä, 2019, 88 p. (+ included articles)
(JYU Dissertations
ISSN 2489-9003; 132)
ISBN 978-951-39-7854-9 (PDF)
Finnish summary
Diss.

As malware continue to evolve, so do the countermeasures which attempt to fight them. A modern computer system typically has many security services installed on top of its operating system which include antivirus, application-control, IDS, firewall, and many more.

Modern operating systems are a highly complex pieces of software which typically contains millions of lines of code. Furthermore, primarily due to endless hardware support, new code is regularly added, resulting in a security sink with an open drain.

Most security services run on top of the operating system and, therefore, are subject to the security of the operating system and its applications. In case of a vulnerability, these services can be removed, thus rendering them them completely useless. This thesis proposes a thin hypervisor-based architecture for a system on top of which a variety of security services can be implemented. These services run in a secure, isolated environment. Furthermore, the proposed system can hide the presence of these security services. The proposed system architecture provides strong security guarantees.

The thesis presents four common, heavily researched security problems and proposes four solutions, which are all based on the proposed architecture. The proposed solutions can compete, and even outperform current solutions, both in terms of security and performance.

Keywords: trusted computing, virtualization, hypervisor, thin hypervisor, unauthorised execution, malware analysis, code encryption, memory forensics

TIIVISTELMÄ (ABSTRACT IN FINNISH)

Leon, Roee Shimon
Hyperviisoreiden sovelluksia tietoturvassa
Jyväskylä: University of Jyväskylä, 2019, 88 s. (+artikkelit)
(JYU Dissertations
ISSN 2489-9003; 132)
ISBN 978-951-39-7854-9 (PDF)

Haittaohjelmien kehittyessä, myös vastatoimet niitä vastaan kehittyvät yhä kiihtyvällä tahdilla. Nykyaikaisessa tietokonejärjestelmässä on tyypillisesti käyttöjärjestelmän päälle asennettu useita tietoturvapalveluita, jotka sisältävät esimerkiksi virustentorjunta-, sovellusohjaus-, IDS-, palomuuri ja muita suojausmekanismeja.

Nykyaikaiset käyttöjärjestelmät ovat erittäin monimutkaisia ohjelmistopaketteja, jotka sisältävät tyypillisesti miljoonia koodirivejä. Lisäksi, pääasiassa valtavan laitteistotuen vuoksi, uutta koodia lisätään säännöllisesti, mistä seuraa tietoturvariskejä.

Useimmat tietoturvapalvelut toimivat käyttöjärjestelmän päällä, ja siksi ne ovat käyttöjärjestelmän ja sen sovellusten turvallisuuden osia. Haavoittuvuuden vuoksi nämä palvelut voidaan poistaa, jolloin ne eivät tietenkään ole käytössä ja ovat siten täysin hyödyttömiä. Tämä opinnäyte ehdottaa ohuita hypervisoripohjaisia arkkitehtuureja järjestelmälle, jonka päälle voidaan toteuttaa erilaisia turvallisuuspalveluita. Nämä palvelut toimivat turvallisessa, eristetyssä ympäristössä. Lisäksi ehdotettu järjestelmä voi piilottaa näiden tietoturvapalvelujen näkyvyyden. Ehdotettu järjestelmäarkkitehtuuri tarjoaa siten vahvat turvallisuustakuut.

Opinnäytetyössä esitetään neljä yleistä tutkittua turvallisuusongelmaa ja ehdotetaan niille neljää ratkaisuvaihtoehtoa, jotka kaikki perustuvat työssä suunniteltuun arkkitehtuuriin. Ehdotetut ratkaisut voivat kilpailla ja toimia tehokkaammin kuin nykyiset ratkaisut sekä turvallisuuden että suorituskyvyn kannalta.

Avainsanat: luotettava tietojenkäsittely, virtualisointi, hypervisori (virtuaalikonemonitori), luvattomat suoritukset, haittaohjelman analyysi, koodin salaus, muistin analyysi

Author	Roee Shimon Leon Faculty of Information Technology University of Jyväskylä Finland
Supervisors	Professor Pekka Neittaanmäki Faculty of Information Technology University of Jyväskylä Finland
	Doctor Nezer Zaidenberg Faculty of Information Technology University of Jyväskylä Finland
Reviewers	Professor Vincenzo Piuri Dept. of Computer Science University of Milan Italy
	Associate Professor, Dr. Jalil Boukhobza Lab-STICC Lab. / Dept. of Computer Science University of Western Brittany France
Opponent	Associate Professor, Dr. Miguel Correia Dept. of Informatics University of Lisbon Portugal

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude to my supervisors, Prof. Pekka Neittaanmäki and Docent Nezer Jacob Zaidenberg, for their guidance and support throughout the process of this thesis.

In addition, I would like to thank my external reviewers for their valuable insights. I am also immensely grateful to Michael Kiperberg, Anat Anatey Leon Zabag, Amit Resh, and Asaf Algawi for co-authoring the joint publications included in this thesis.

I want to thank the COMAS Graduate School, which partially funded this research. Furthermore, I would like to thank the University of Jyväskylä staff for being so responsive and helpful during my years as a Ph.D. student.

I wish to thank my beloved wife, Anat Anatey, without whom this thesis would not have been possible, for her continuous moral support, co-authoring, and help.

Finally, I want to thank my parents, Anat and Eyal, for their love and support throughout my entire life, not least during my work on this thesis.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

LIST OF INCLUDED ARTICLES

PI R. Leon, M. Kiperberg, A.A. Leon Zabag, A. Resh, A. Algawi, N.J. Zaidenberg. Hypervisor-Based White Listing of Executables. *IEEE Security & Privacy, Vol. 11, No. 5., pp. 58-67*, 2019.

PII M. Kiperberg, R. Leon, A. Resh, A. Algawi, N.J. Zaidenberg. Hypervisor-assisted Atomic Memory Acquisition in Modern Systems. *ICISSP Conference*, 2019.

PIII M. Kiperberg, R. Leon, A. Resh, A. Algawi, N.J. Zaidenberg. Hypervisor-based Protection of Code. *IEEE Transactions on Information Forensics and Security, Vol. 4, No. 8, pp. 2203-2216*, 2019.

PIV A. Resh, M. Kiperberg, R. Leon, N.J. Zaidenberg. Preventing Execution of Unauthorized Native-Code Software. *International Journal of Digital Content Technology and its Applications (JDCTA), Vol. 11, No. 3, pp. 72-90*, 2017.

PV A. Resh, M. Kiperberg, R. Leon, N.J. Zaidenberg. System for Executing Encrypted Native Programs. *International Journal of Digital Content Technology and its Applications (JDCTA), Vol. 11, No. 3, pp. 56-71*, 2017.

PVI R. Leon, M. Kiperberg, A.A. Leon Zabag, N.J. Zaidenberg. Hypervisor-assisted Dynamic Malware Analysis. *ACM Transactions on Privacy and Security (TOPS), Submitted*.

The idea of a more efficient encrypted code execution using the buffered execution technique, described in article [PV] [RKL+17b], was devised by the author together with A. Resh, M. Kiperberg, and N. Zaidenberg, which improved their previous work [AKZ13] in terms of security. The author designed and implemented the decryption system, as well as performing and describing the performance analysis in article [PV].

The whitelist executable database introduced in article [PIV] [RKL+17] was implemented by the author. In addition, the design of the system was devised by the author together with A. Resh, M. Kiperberg, and N. Zaidenberg. The author performed and described the performance analysis in article [PIV].

Article [PIII] [KLR+19] describes an encrypted code execution method that is more secure and more efficient than the one described in [PV]. The author implemented the secure key-exchange with the TPM and parts of the hypervisor just-in-time decryption. Furthermore, the complete evaluation and analysis were conducted by the author.

Articles [PI] [LKLZ+19] and [PVI], which were mainly written by the author, present two hypervisor-based solutions to common security problems. In [PI], the author implemented a hypervisor-based unauthorized execution prevention system.

In Paper [PII] [KLR+19b], the author used virtualization for forensics and security. In paper [PVI], the author implemented a dynamic hypervisor-based malware analysis system. The co-authors presented, as well as assisting in writing, the evaluation of the solutions.

The author has also authored and participated in the following research papers (not included in this thesis):

1. The author researched ARM security alternatives which resulted in a conference paper [AKL+19] and a book chapter [ZBYL20]. The co-authors have ported [PIII] to the ARM platform using *hyplets* [BYZ18], which resulted in [BYZ19].

2. The author has participated in authoring hypervisor-based protection against Structured Exception Handler (*SEH*) attacks [AKL+19b]. A unique use case of Paper [PI] in a Virtual Desktop Infrastructure (VDI) is described in [AKL+19c].

3. While researching forensics solution for Paper [PII], the author researched how a hypervisor can protect itself from detection. The author has contributed to [AKL+19] conference paper and [AKL+20] book chapter.

1 INTRODUCTION

Computer security, or cybersecurity, is a critical issue. Ransomware, Viruses, Trojans, and Rootkits are just a few of the threats that the computer industry faces daily [W03]. In the face of these threats, the computer security industry offers a variety of defences (e.g., antiviruses, firewalls, IDSes), and secure coding standards [LMS+11, S08] have been introduced to software developers to avoid security incidents in the future. However, even as the computer security industry evolves to provide better solutions to known threats, attackers continue to upgrade their attacks to exploit the weaknesses of these solutions.

A core component of almost every computer system is the operating system and its applications. The operating system is responsible for managing the hardware of the computer system as well as for providing services to programs. Due to the high variance between hardware devices and the rich set of services provided to applications, a typical operating system (e.g., Linux) is complex and may consist of millions of lines of code [JO05]. To reduce the amount of code that is required to execute with kernel privileges, a concept of microkernel was introduced [DPM02]. A pure microkernel contains the bare minimum software required to implement an operating system. The rest of the operating system components are implemented as user applications. Despite the security advantages of microkernels, the most popular operating systems are either monolithic or hybrid, primarily due to performance reasons. As a consequence, the attack-surface of such systems is broad, and zero-days vulnerabilities are often found.

Most of the security services available today come in the form of a user application or a kernel driver. In both cases, these services provide weak security guarantees, as vulnerabilities within the operating system (and its services) may be used to remove the software [AD10], hence effectively removing the protection. Furthermore, due to the complexity of modern operating systems, adding a security service on top of an existing operating system may be a non-trivial task.

Security services, which are both persistent and transparent can be provided using a thin hypervisor [SLQ07, SET+09, SK10, WJ10, Z18, KLR+19, LKLZ+19]. This thesis presents the applications that arise from using a thin hypervisor for security purposes. In particular, the security services presented herein provide strong security guarantees and can be deployed on a machine without any modifications of the running operating system. Finally, beyond being more secure, these services also, in most cases, outperform current solutions.

1.1 Modern Operating Systems

1.1.1 Security of Modern Operating Systems

An operating system typically controls a computer system. Modern operating systems are incredibly complex, and an example of a complex modern operating system component is the graphical user interface (GUI). A typical GUI is composed of multiple layers, each of which is a separate application with a different role. This design is highly modular; however, the user input must go through all of these layers before the actual hardware processes it. In case of a vulnerability in any of these layers, attacks such as spoofing and data theft are possible [BCI+15]. Furthermore, these GUI systems often do not provide isolation between applications [SVN+04]. Due to their complexity, modern operating systems are more prone to security vulnerabilities [R06, AD10].

Nevertheless, naturally, most of the security services available today run on top of the operating system, either as a user application or a kernel driver. Consequently, these applications share resources (and sometimes even address space) with other operating system components and applications, and a flaw in any of these applications or components may compromise the entire system [L11].

1.1.2 Limitations of Kernel Applications for Security Purposes

Security services that are implemented as user applications are quite limited. For example, to implement a firewall, the software may need to intercept every packet before it is sent or received by the operating system. In case of a rule violation, the software rejects the packet; otherwise, it allows it to be sent or received. The latter requires intervention in the operating system packet processing process and therefore, can be only implemented as a kernel driver. For the sake of packet filtering, for example, Microsoft Windows provides the *Windows Filtering Platform* (WFP), and similar packet-filtering capabilities are also available in Linux. However, formal APIs are not available for all needs, in particular, for security needs. For example, malware analysis software may need to intercept every single system call conducted by a monitored process for the sake of behaviour inspection. Microsoft Windows, for example, does not provide such mechanisms, and consequently, the software must perform one of two tasks: (1) modify the operating system code or data such that every system call is intercepted by the software, or (2) directly modify application API code (e.g., *ntdll.dll/kernel32.dll* in Windows) that performs the actual system calls. The second option is indeed possible and will work properly on most operating systems. However, it has several deficiencies (see Paper [PVI]). The first option requires modifications of the operating system code or data.

Due to its complexity, modifications of the operating system code or data without using formal APIs is a difficult and risky task that may harm the stability and security of the operating system. For example, one may modify operating system data that is supposed to be guarded with a lock without locking or checking the lock first. In 2005, Microsoft introduced *Kernel Patch Protection KPP* (also known as PatchGuard) [UninformedKPP], the goal of which is to prevent operating system code or data modification attempts. All 64-bit versions of Microsoft Windows are now shipped with *KPP*. Due to *KPP*, even legitimate modifications conducted by kernel drivers to operating system code or data are not permitted.

1.2 Common Security Problems

Many challenging problems exist in regard to cybersecurity, some of are heavily researched. Examples of such challenges are:

1. Data protection – protection of data from being stolen or tampered [SER12].
2. Memory analysis – analysis of a memory image for the purpose of cyber crime investigation [VF11].
3. Malware analysis – analysis of malware for the purpose of determining its impact on a computer system [GBS14].
4. Network forensics – analysis of network traffic for the purpose of detecting potentially malicious behaviour [PJN10].

For most of the problems, both hardware and software solutions exist. Hardware solutions typically involve dedicated hardware specifically designed to solve or assist in solving a specific security problem. Software solutions typically involve user-space or kernel-space applications designed to solve a specific security problem, and they are typically installed on top of the running operating system.

The author researched four heavily researched security problems: (1) Memory forensics, (2) Malware analysis, (3) Unauthorized execution prevention, and (4) Preventing reverse engineering of software. The following subsections briefly describe the background to these problems.

1.2.1 Memory Forensics

Memory forensics is the analysis of a computer physical memory dump primarily for cyber-crime investigation. If the memory dump is reliable, much useful information regarding the current state of the system can be extracted. For example, a memory forensics tool that is familiar with the internal operating system data structures can extract the list of active processes, opened files, and more.

The problem of memory forensics can be broken up into two phases: (1) acquisition of the physical memory and (2) static analysis of the physical memory dump. Plenty of practical static analysis tools exist for phase two (e.g., [C14, PWF+06]). A challenging task regarding physical memory acquisition is atomicity. To better understand the problem of atomicity, consider a situation on which pages 1,000-1,500 are used by the operating system for internal dynamic allocations, and consider too that the first 1,500 pages have already been dumped. The operating system creates an internal data structure (e.g., *PCB*) node that is located on page 1,200 and updates the data-structure on page 3,000 (not yet been dumped) to point to the newly created node. As a result, the dumped data structure is not in a valid state. In many cases, these potential inconsistencies may result in incorrect analysis. Therefore, memory acquisition solutions must take special measures to assure a consistent memory image.

1.2.2 Malware Analysis

Malware analysis is the process of examining the behaviour of a given malware. The analysis process can assist the malware analysts to understand the malware's means of operation as well as providing countermeasures against similar malware.

There are two ways to perform malware analysis: (1) using static methods and (2) using dynamic methods. Static methods typically involve reverse engineering of the malware while dynamic methods typically involve running the malware and observing its behaviour. The main advantage of the static methods is that the malware does not need to be executed; thus, no damage can be done to the machine on which the malware is executed. However, modern malware use an abundance of evasion techniques [YY10, OSM11, RMI12, BLS13] to make static analysis difficult. Therefore, though it requires execution of the malware, dynamic analysis is often the preferred method.

1.2.3 Unauthorised Execution Prevention

The problem of unauthorised code execution is one of the most basic computer security problems. Vulnerabilities due to memory overflows, and even phishing techniques, can be used to achieve unauthorised code execution. Determining whether a given code is authorised for execution is an extremely challenging task that preoccupies many researchers. Idika, N. et al. [IM07] have divided current solutions into three categories: behavioural, signature-oriented, and whitelist-oriented. Solutions that are behavioural are difficult to evaluate, as these solutions are based on heuristics. Signature-oriented solutions are prone to zero-day attacks, as they are based on prior knowledge of known attacks. Whitelist-based systems contain a database of allowed executables. Only executable images that exist in the whitelist-database are allowed to run. Therefore, whitelist-based systems provide the strongest security guarantees [M09]. However, these systems are also the most restrictive. Nonetheless, in environments that do not tend to change frequently, the preferred option is a whitelist-based system.

1.2.4 Preventing Reverse Engineering of Software

Protecting software from theft, illegal use, and even attacks can be done by protecting it against reverse engineering. One of the most common methods with which to achieve the latter is obfuscation. In obfuscation, the code of the program is translated into a semantically identical, but much harder to understand, code. An example of an obfuscation technique is instruction substitution, in which original instructions are replaced with semantically identical instructions. Obfuscation has, however, been proven to be ineffective [ROL09]. An effective method for protecting a program against reverse engineering is encryption. However, two issues arise when encryption is considered:

- Key storage – the decryption must be stored in a location accessible only to the protected software.
- Decrypted code storage – the decrypted code must not be visible to different software during execution.

Though encryption is more effective than obfuscation, the latter remains the most commonly used countermeasure to protect a program from reverse engineering.

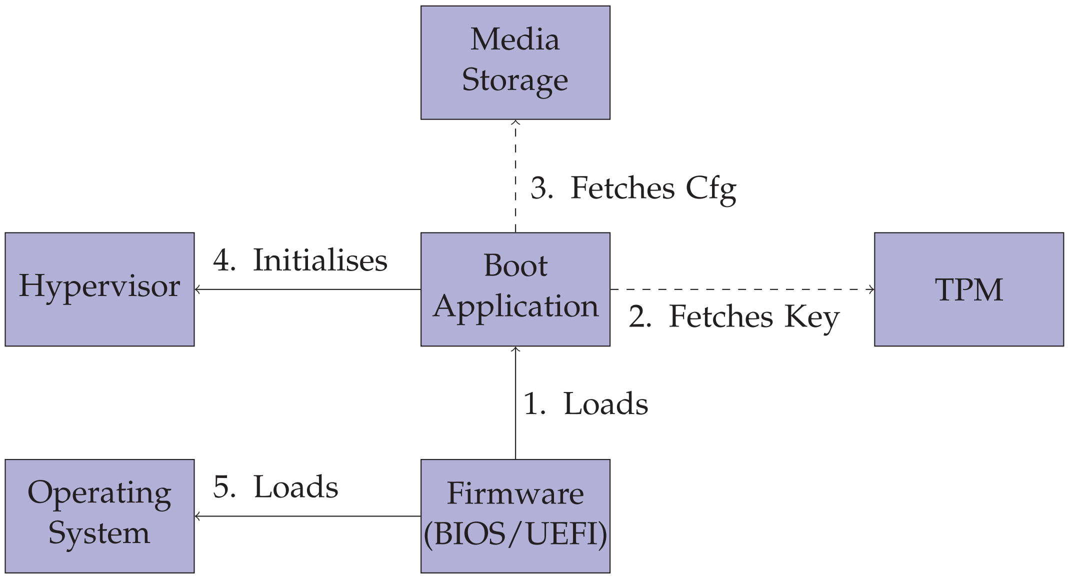


FIGURE 1 The common architecture of our method. The dashed arrow indicates an optional step. The solid arrow indicates a mandatory step.

1.3 Our Method

This thesis provides a detailed description of hypervisor-based solutions to the common security problems described in the previous section. The solutions presented herein have a common architecture that can be used to solve other security problems. The structure of this chapter is as follows. First, we briefly describe the common architecture of the solution before turning to how our method is used to solve each of the common security problems described above.

1.3.1 Common Architecture

The common architecture of our method is composed of several components, some of which are optional. Figure 1 depicts the components and the relations between them. First, a boot application is loaded by the firmware. Then, optionally, the boot application fetches a cryptographic key from the *TPM* and a module (i.e., a security service) specific configuration from a storage media (e.g., a disk-on-key). Next, the boot application initialises a hypervisor, which retrieves the *TPM* data and the configuration as parameters from the boot application. Finally, the boot application returns the execution control to the firmware, which in turn loads the operating system. The hypervisor remains active and in memory.

1.3.2 Memory Forensics

Atomic memory acquisition is not a trivial task in modern systems. The memory to be acquired needs to be locked for write access during the dump process. Furthermore, it may be accessed by more than one processor at a given time, making it difficult to ensure atomicity. A hypervisor can be used to ensure that write access to a physical page is prohibited until the page has been successfully dumped [MFP+10, QXM17]. However, write access to some of the operating system pages cannot be blocked, as the access time to these pages is critical (e.g., pages that are accessed during initial interrupt handling). Moreover, the location of these pages cannot be easily determined due to *ASLR*, which is implemented in most modern operating systems.

Our method can obtain an atomic memory image of a running system. Furthermore, our method supports modern operating systems, in particular, those that implement *ASLR* and support multiprocessor architecture. The architecture of our method consists of the components described in Section 1.3.1 except for a *TPM*, which is not required. During initialisation, the boot application fetches a configuration file that contains information that allows the hypervisor to locate the operating system internal data structures, regardless of *ASLR*. The hypervisor uses these internal data structures to obtain information regarding the physical pages that cannot be blocked for writing. The hypervisor takes the following steps to obtain an atomic memory image. First, the hypervisor ensures that all processors have no write access to the physical memory. Next, the hypervisor writes the pages that cannot be blocked for writing to a pre-allocated buffer. Finally, the hypervisor lazily dumps the rest of the physical memory.

1.3.3 Malware Analysis

Sophisticated malware use an abundance of evasion techniques to avoid detection by static analysis tools. One such technique is obfuscation, which allows the malware to camouflage itself from detection, for example, by including encoded or encrypted instructions which are only decoded or decrypted during execution [YY10, OSM11, RMI12, BLS13]. In a dynamic approach, the malware is executed (typically, in an emulated or virtualized environment) to be analysed. Current dynamic malware analysis methods can be divided into four classes: (1) hooking methods, (2) emulation methods, (3) hypervisor-based methods, and (4) bare-metal-based methods. Hooking methods [WHF07, GF10, YMH+17] perform in-line code modifications and are therefore detectable. Emulation methods, e.g., [BKK06], are slow and may be exploited due to incomplete emulation [F06, F07, RKK07, DRS+08, VC14]. Hypervisor-based methods [DRS+08, LMP+14] provide a malware analysis system that is both transparent (i.e., performs no modifications to the running operating system) and is more efficient compared to emulation methods. Both [DRS+08] and [LMP+14] are built on top of the XEN hypervisor [BDF+03] and consequently run the malware sample inside a regular domain, which remains untouched, while the actual analysis takes place in the control domain (dom0). The latter modus operandi forces multiple transitions for each event (e.g., a system call), and therefore induces a significant performance overhead. Bare-metal-based methods are limited as they only track activities that can be externally monitored (e.g., network and disk activities).

Our method can be classified as a hypervisor-based method. However, in our method:

1. The system call monitoring component is located within the guest operating system and consequently no intervention of the hypervisor is required during the system call monitoring and analysis. As a result, the performance overhead is kept to a minimum.

2. Though completely handled in the guest context, the monitoring component is fully transparent to the guest operating system.

The architecture of our method consists of the components described in Section 1.3.1 except for a *TPM*, which is not required. During initialisation, the boot application fetches a configuration file that indicates how the system calls induced by the malware are to be recorded. The hypervisor installs an operating system monitoring component that performs the actual system call hooking. The hypervisor ensures that the monitoring component remains undetected.

1.3.4 Unauthorised Execution Prevention

Malware detection is a heavily researched topic, not only because of its importance but because malware continue to evolve even as malware detection methods develop. Researchers have divided current malware detection techniques into three categories [IM07]. Paper [PI] describes these categories in detail. Whitelist-based systems, which are typically the preferred option in systems that do not tend to change frequently, suffer from several deficiencies: (1) They are prone to vulnerabilities in the operating system and (2) they check whether an executable image is valid only at the time as it is loaded into memory.

Our method can be categorised as a whitelist-based method and is associated with two main advantages over current methods:

- In our method, the execution of a given executable image is enforced during the executable's entire lifetime.

- Our method is not prone to vulnerabilities in the operating system and can block unauthorised execution even if the operating system kernel itself is compromised.

The architecture of our method consists of the components described in Section 1.3.1 except for a *TPM*, which is not required. In addition to these components, our method also consists of a whitelist executable scanner. The whitelist executable scanner is responsible for creating a whitelist database which is used by the hypervisor to enforce authorised execution. During initialisation, the boot application fetches a configuration file which contains the whitelist database. The hypervisor then sets the entire physical memory pages to have no execute rights. Upon an execution attempt, the hypervisor checks whether the page to be executed is valid against the whitelist database. Though providing real-time page granularity protection, our method induces only a negligible performance overhead.

1.3.5 Preventing Reverse Engineering of Software

A compiled program is susceptible to reverse engineering attacks on its algorithms and business logic. Current countermeasures against reverse engineering of code are based on obfuscation. However, obfuscation methods can be broken with sufficient effort [ROL09]. Furthermore, the more sophisticated the obfuscation, the more performance overhead is induced [Paper PII] [JRW+15]. A better code protection technique is encoding which either encrypts or compresses portions of the original program, which are later decrypted or decompressed before their execution. Unfortunately, these methods embed their cryptographic key within the actual algorithm and are therefore susceptible to automatic code extraction attacks [R17].

Our method can be categorised as an encoding method. However, our method provides two main advantages over current methods:

- In our method, the decryption key is not embedded in the algorithm. Instead, the key is embedded in a *TPM*.
- In our method, the decrypted code is protected even from an attacker that has full access to the operating system kernel and can sniff on any bus.

The architecture of our method consists of the components described in Section 1.3.1, including a *TPM*. Furthermore, our system also consists of an encryption tool. The encryption tool is responsible for encrypting a program. The hypervisor later decrypts the encrypted code to prepare it for execution. During initialisation, the boot application fetches a cryptographic key from the *TPM*, which is only possible if the machine is in a valid state, and a configuration file that contains the encrypted functions. The hypervisor stores the cryptographic key and the decrypted functions in a special memory region that never evicts from the cache. Upon an attempt to execute an encrypted function, the hypervisor decrypts the function and executes it on its behalf.

2 HYPERVISORS

2.1 Virtualization

The need to effectively share hardware resources amongst a group of users was apparent even in the 1960s. It was the expensive computer hardware, high maintenance costs, and inefficient hardware use, which led to the need for virtualization. Virtualization allows for the physical hardware resources to be divided amongst multiple virtual devices. The most common application of virtualization is running multiple operating systems on top of the same hardware. The latter allows for more efficient resource use and is much easier to manage. The software that manages these virtual machines is typically referred to as the *VMM* or hypervisor. The hypervisor is referred to as the *host* while the virtual machines it manages are referred to as *guests*. Hypervisors were also used for other purposes [BS96, RK07, KZA11, KZA+13]. There are two main types of virtualization: *full-virtualization* and *para-virtualization* [NC05, YHB+11]. In para-virtualization, the virtualized operating system is aware that it is being virtualized: That is, whenever a hardware resource needs to be accessed, it explicitly calls the hypervisor (via a predefined *API*). Therefore, the hypervisor does not need to simulate the hardware resources. In full-virtualization, the virtualized operating system is not aware that it is being virtualized, and, therefore, does not require any modifications. Full-virtualization software can be software-assisted or hardware-assisted (if appropriate hardware is available). Software-assisted full-virtualization can be implemented using binary translation techniques. Binary translation techniques involves re-writing of the guest operating system and is often slower than modern hardware-assisted virtualization implementations.

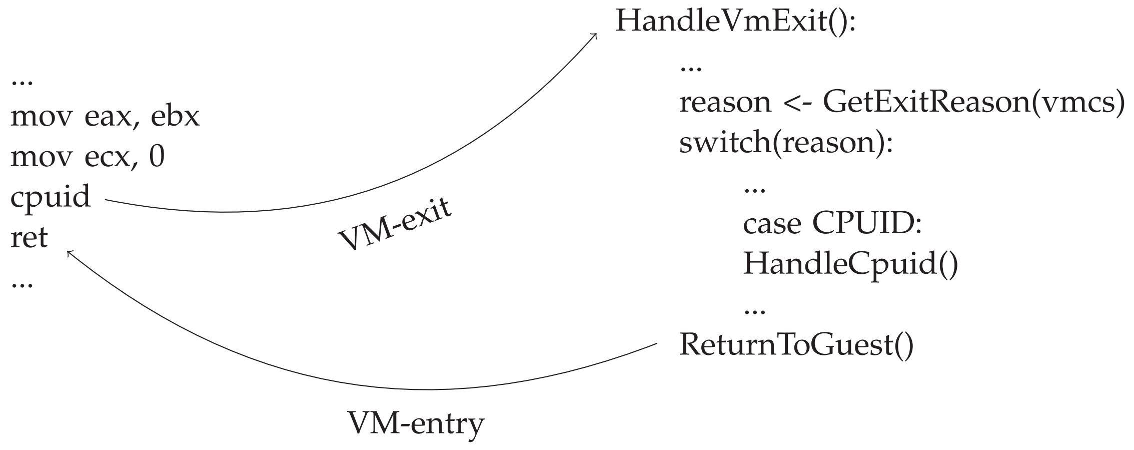
2.2 Hardware-assisted Virtualization

Hardware-assisted virtualization effectively eliminates the need for binary translation. Instead, the processor is equipped with special hardware that allows it to intercept certain hardware-related events. Both Intel [INTEL16] and AMD [AMD18] have integrated hardware-assisted virtualization into their *x86* processors since 2005. Virtualization extensions for *ARM* processors have been added to ARMv7-A architecture in 2010 [ARM10]. In this work, we discuss Intel implementation for hardware-assisted virtualization.

A special structure referred to by Intel as the Virtual Machine Control Structure (VMCS) is defined for each virtual machine managed by the VMM. This control structure is logically separated into three parts: (1) host fields, (2) guest fields, and (3) control fields. The host fields define the context of the machine while the host is running. The guest fields define the context of the machine while the guest is running. This context includes the location of the page tables, interrupt descriptor table, and many additional values that constitute the internal state of the virtual machine. The control fields allow the VMM to define the events that trigger a transition from the guest to the hypervisor. The events may be synchronous (e.g., execution of `INVLPG` instruction), or asynchronous (e.g., page-fault exception). The transition from the guest to the hypervisor is referred to as VM-exit, while the transition from the hypervisor back to the guest is referred to as VM-entry. After a VM-exit, the processor executes in a special mode Intel refer to as VMX root-mode (i.e., host privileges). After a VM-entry, the processor executes in VMX non-root mode (i.e., guest privileges). Some events must be intercepted, while others cannot be intercepted. Example of an event that always causes a VM-exit is the `CPUID` instruction. By contrast, the `SYSENTER` and `SYSCALL` instructions cannot be configured to cause a VM-exit. The VMM event-handler must be configured by the hypervisor and is a part of the VMCS. This handler is also referred to as the VM-exit handler. Figure 2 depicts the transition process from the hypervisor to the guest and vice versa.

A technology called Second Level Address Translation (SLAT), or Extended Page Tables (EPT) in Intel terminology, allows mapping between the physical memory as it perceived by the guest and the actual physical memory to be defined. This capability is somewhat similar to the concept of virtual memory in operating systems in which the operating system can define the mapping between the memory as it perceived by the process (i.e., the virtual memory) and the physical memory. Similarly to virtual page tables, the EPT allows the hypervisor to specify the access rights of each guest physical page. In case the guest attempts to access a physical page that is either unmapped or has insufficient access rights, an event called EPT-violation occurs; triggering a VM-exit.

Input-Output Memory Management Unit (IOMMU), or Virtualization Technology for Directed I/O (VT-d) in Intel terminology [IntelIM18], allows the hypervisor to specify the mapping of the physical address space as perceived by the hardware devices to the real physical address space. It is a complementary tech-



```
...
mov eax, ebx
mov ecx, 0
cpuid
ret
...
```

```
HandleVmExit():
    ...
    reason <- GetExitReason(vmcs)
    switch(reason):
        ...
        case CPUID:
        HandleCpuid()
        ...
    ReturnToGuest()
```

FIGURE 2 Transition from guest to host and vice versa. The processor executes the `CPUID` instruction while in the context of the guest. As a result, a VM-exit occurs, and the execution control is transferred to the handler configured by the VMM (HandleVmExit). Next, the handler determines the cause of the reason and handles it (HandleCpuid). Finally, the handler issues a VM-entry (i.e., continues execution of the guest).

nology to the EPT that allows a construction of a coherent guest physical address space for both the operating system and the devices.

2.3 Thin Hypervisor

Full hypervisors like *XEN* [BDF+03], *VMware Workstation*, and *Oracle VirtualBox* can execute multiple operating systems concurrently. The main purpose of the hardware virtualization technology is to provide means with which software developers can implement full hypervisors. In this work, we propose to use a special breed of hypervisor, a thin hypervisor, which runs only a single operating system. In contrast to full hypervisors, a thin hypervisor is used to provide security services to the guest operating system. Thin hypervisors have been used for security purposes by many researchers [SLQ07, SET+09, WJ10, SK10, Z18, LKLZ+19, KLR+19, ZKBY+]. Security services that are thin hypervisor based have two major advantages over operating system kernel based security services: (1) They run in an isolated environment and (2) they run in a higher privilege mode. In a sense, a thin hypervisor is more similar to the *ARM Security Extensions* [ARM03] technology, which was also been used by many researchers [XHT07, SSW+14, YMH+17, YMM+17, ZBYL20], than a full hypervisor. A thin hypervisor typically intercepts only a (typically small) set of events that allows it to provide a security service. Most of the hardware related events are handled directly by the guest operating system. Figure 3 depicts a thin hypervisor supporting a single guest. A thin hypervisor must also protect itself from subversion. Consider the following simple attack as an example. A hypervisor has set the guest physical memory to be directly mapped to the real physical memory.

An attacker with kernel privileges could map the physical pages on which the hypervisor resides and then modify the VM-exit handler as desired. The same attack can be carried out using a malicious I/O device [AD10].

The steps taken by the hypervisor to facilitate a secure environment varies by the security service type. However, two steps are common for all services:

1. The hypervisor must set the machine such that the memory on which it resides is inaccessible to the processor while it executes in non-root mode.
2. The hypervisor must set the machine such that the memory on which it resides is inaccessible to hardware devices via Direct Memory Access (DMA).

A properly configured hypervisor can provide strong security guarantees.

In some cases, a thin hypervisor must also protect itself (and its actions) from detection. An example of such a case is malware that alters its behaviour if it detects the presence of a hypervisor. Theoretically, a guest operating system is unaware of the hypervisor's intervention, as it experiences a normal hardware access cycle. However, the hypervisor mediation has a time-toll, and it may cause processor side effects. These properties have led to a debate regarding the detectability of a hypervisor [FLM+07, GAW+07, RT07, RT08, BYDD+10, BBN12].

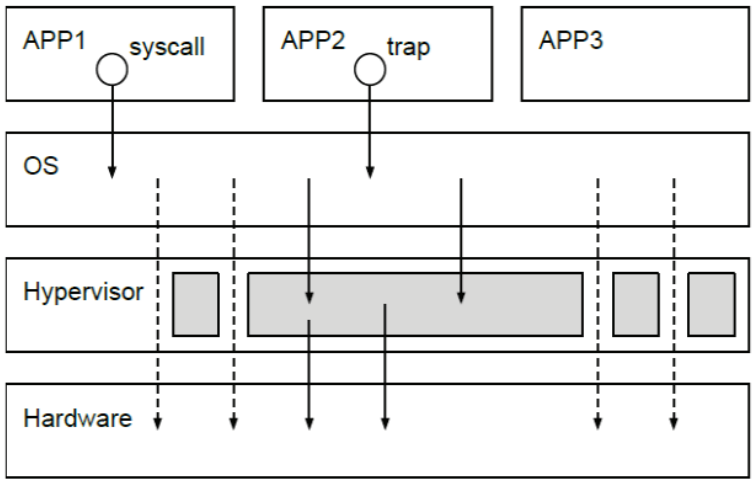


FIGURE 3 A thin hypervisor. The hypervisor runs on a higher privilege level than the operating system. System calls, traps, exceptions, and other interrupts transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting services from the underlying hardware. A thin hypervisor can intercept some of those requests and handle them according to some policy.

2.4 Thin Hypervisor Boot and Initialisation

Popek G. and Goldberg R. [PG74] have classified two types of hypervisors: (1) type 1 (or bare-metal) hypervisors and (2) type 2 (or hosted) hypervisors. Type 1 hypervisors are operating systems designed for virtualization purpose, examples of which are *VMware ESXI* [VMware] and *XEN* [BDF+03]. Type 2 hypervisors are hosted on a general-purpose operating system and consist at minimum of a management application (which runs as a user process) that allows for the creation or modification of virtual machines, and a kernel driver that initialises a hypervisor. In contrast to type 1 hypervisors, type 2 hypervisors use operating system services to access the hardware and therefore are, somewhat inaccurately, considered to be running on top of an operating system. Examples of type 2 hypervisors are *VMware Work Station* and *Oracle VirtualBox*. The main advantage of type 1 hypervisors is their performance and security, while the main advantage of type 2 hypervisors is the simplicity with which they are implemented and their driver support (the operating system kernel provides these).

The classification described in the first paragraph is less suitable for thin hypervisors for two prominent reasons:

- A thin hypervisor is usually designed for security purposes. Therefore, using existing operating system functionality is often a bad idea because in most cases, the threat model states that the operating system cannot be trusted. Consequently, the main advantage of the type 2 hypervisor becomes insignificant.

- A thin hypervisor does not run multiple operating systems but rather a single guest.

Security researchers have introduced a type 2 thin hypervisor that functions as an undetectable malware [RT06B]. This malware initialises a hypervisor on the target processor and transforms the running operating system into a guest.

Implementation of security services using a type 2 thin hypervisor is more difficult to implement because:

- The initialisation of the hypervisor must occur while the operating system is running. For example, Intel's Input-output Memory Management Unit (IOMMU) [IntelIM18] must be initialised at a deterministic transaction boundary, making it difficult for a hypervisor to reinitialise. Furthermore, the hypervisor must be authenticated before deployment (see Papers [PIV, PV]).

- Part of its implementation is operating system dependent (implemented as a kernel driver).

In our method, we use a variation of a type 1 hypervisor. First, a boot application is loaded by the firmware. Next, the boot application performs some preparations and initialises a hypervisor. Then, the boot application returns to the firmware, at which point the hypervisor has control over the system and the firmware runs as a guest. Finally, the firmware loads the original boot application (typically, the operating system boot loader). The hypervisor remains active and in memory.

2.4.1 BIOS Boot

In a BIOS enabled systems, during the boot process, the firmware attempts to locate a bootable device according to the defined boot order (typically stored on a CMOS chip). A bootable device is identified by a special sector, called the Master Boot Record (MBR), at its very beginning. The MBR device contains up to four partitions, of which one must be marked as active (typically, the one which has the operating system first level bootloader). Once an active partition is found, the BIOS loads its first sector to a pre-defined location and begins to execute it.

The hypervisor boot application must first be installed to a bootable storage device. Next, the partition on which the hypervisor boot application was installed must be marked as active, after which the machine must be rebooted. Finally, the boot application initialises a hypervisor and returns to the original operating system boot loader.

2.4.2 UEFI Boot

In a UEFI [UEFI] enabled systems, after a successful initialisation, the firmware loads a sequence of executable images, called UEFI applications. The sequence is stored in a firmware-defined non-volatile storage. The partition on which a UEFI application is stored must have a FAT32 file system and is typically referred to as an EFI System Partition (ESP). Each element in the sequence points to a location that contains UEFI application. After loading a UEFI application to the memory, the firmware calls the application *main* function. If the *main* function returns, the firmware loads the next application and so on. Typically, the operating system's boot loader is implemented as a UEFI application, whose *main* function does not return. The firmware settings screen allows the boot sequence to be configured.

'Secure boot' is a feature provided by UEFI which allows a computer system owner to authenticate UEFI applications before their execution, thereby protecting the executable image from malicious modifications.

The hypervisor boot application must first be installed to a storage device, on an ESP partition. Next, the boot sequence must be altered such that this partition is the first in the sequence. Optionally, 'secure boot' can be configured, after which the machine needs to be rebooted. Finally, the boot application initialises a hypervisor and returns to the UEFI firmware, and the latter loads the next UEFI application in the sequence (typically, the operating system bootloader).

2.4.3 Initialisation

The firmware stores the boot application in a reclaimable memory location. That is, the firmware reports this memory location as available to the operating system. Because the hypervisor must remain active during the entire lifetime of the system, memory allocated by the boot application must be persistent (not reclaimable). In UEFI enabled systems, persistent memory can be allocated using UEFI memory allocation functions. In BIOS enabled systems, the list of available memory ranges can be obtained by using the `INT15h` instruction and specifying the value *e820h* in the *ax* register. Simply put, software requests the BIOS for the next memory range until no more ranges are available (similar to a linked list). The boot application must modify these memory ranges to allocate persistent (not reclaimable) memory. Algorithm 1 describes two procedures. The first procedure is part of the boot application and is responsible for memory allocations. First, the original memory map is saved to a global variable (SavedMemoryMap)(line 1). The allocation handler iterates the memory map and looks for a usable range that is large enough to hold the allocation (lines 3-4). When such range is found, it subtracts the desired allocation size from it (line 5). Finally, it computes the allocation base address and returns it (lines 6-7). The second procedure is part of the hypervisor and is responsible for the interception of the e820 requests. First, it retrieves the next memory range in the saved (modified) list (line 11) and copies it into a buffer provided by the operating system (line 14). If this is the last range in the memory map, it marks it as such (lines 12-13). Finally, the original instruction is skipped (line 15).

Algorithm 1 BIOS memory allocation and the HV's memory map interception functions

```
 1: $SavedMemoryMap \leftarrow GetOriginalMapFromBios()$
 2: procedure HANDLEBIOSALLOCATION($size$)
 3:     for each $m \in SavedMemoryMap$ do
 4:         if $m$ is marked as usable and $m$ is large enough then
 5:             subtract required size from the length of $m$
 6:             $allocatedBase \leftarrow length(m) + base(m)$
 7:             return $allocatedBase$
 8:     return $NULL$
 9: procedure GETMEMORYMAPINTERCEPT()
10:     $address \leftarrow GetOSOutputBuffer()$
11:     $range \leftarrow GetNextMemoryMap(SavedMemoryMap)$
12:     if range is the last memory range then
13:         Mark range as last
14:     Copy range to address
15:     Skip the instruction
```

Algorithm 2 depicts the initialization process of the hypervisor. First, the boot application allocates the persistent memory required for the hypervisor. This memory must hold its code, data, and internal data structures (e.g., a VMCS for each processor). The boot application copies the hypervisor's code and data to the allocated memory and sets the instruction pointer to the code's location. Next, the boot application retrieves module specific configuration from a storage media. Optionally, the boot application retrieves a cryptographic key from the TPM. Finally, the boot application initialises a hypervisor on each of the available processors and returns to the original operating system boot application.

Algorithm 2 Thin Hypervisor Initialisation

```
 1: Allocate physical memory for the hypervisor
 2: Copy the hypervisor image (code and data) to the allocated memory
 3: Set the instruction pointer to the newly allocated memory
 4: Allocate physical memory for the configuration
 5: Retrieve module specific configuration from a storage media
 6: Copy the retrieved configuration to the allocated memory
 7: Optionally retrieve a cryptographic key from the TPM
 8: for each $p \in \mathcal{P}rocessorsList$ do
 9:     Initialise a hypervisor on processor $p$
10: Return to the original operating system boot application
```

2.5 Thin Hypervisor Security

Secure operation of the hypervisor first and foremost requires a secure boot process. Upon a successful boot and initialisation, the hypervisor must protect itself from subversion. Depending on the provided security service, the hypervisor might also need to protect internal secrets from cache eviction attacks. These security requirements are described in the following subsections.

2.5.1 Secure Boot

A secure boot process implies that the media storage on which the boot application is stored (along with the hypervisor) is left untampered. Consider the following attack scenario. A skilled attacker that can execute code with kernel privileges locates the disk on which the boot application is stored (e.g., a UEFI partition in case of a UEFI boot) and modifies it as he wishes. On the next boot, the tampered boot application will be loaded by the firmware.

'Secure boot' is a feature provided by UEFI that allows a computer system owner to authenticate UEFI applications before their execution, thereby protecting the executable image from malicious modifications. The first phase in the UEFI boot process is the firmware initialisation phase, also called the security (SEC) phase because it serves as the basis for the root of trust. After the SEC phase is completed, the trust is maintained via public key cryptography. Paper [PI] describes in detail how a machine owner can use the UEFI secure boot to ensure the secure boot of the hypervisor's boot application.

In a BIOS-enabled systems, a Trusted Platform Module (TPM) can be used to provide a secure boot. TPM [M11, ZNK+15] is a standard that defines a device with a non-volatile memory and a predefined set of cryptographic functions. The TPM allows remote attestation of the software that executes during the boot process, which is done by recording the hashes of the running images (therefore, also allowing it to be used to enforce boot order). These hashes are placed in a list called the Measurement List (ML), which is securely stored within the host TPM. A remote machine can then be used to check whether the measurement list corresponds to an expected (previously defined) trusted configuration.

2.5.2 Memory Protection

Second level address translation (SLAT), or extended page tables (EPT) according to Intel's terminology, is a mechanism that allows hypervisors to control the mapping of physical page addresses as they are perceived by the guest to the real physical addresses. In this sense, SLAT is similar to virtual page tables, a mechanism that allows the operating system to control the mapping of addresses as they are perceived by the process to the real physical addresses. When a process in a virtualized environment attempts to load a variable at (a virtual) address x, this address is first translated by the virtual page tables to a guest physical address y, then y is translated by EPT to a (real) physical address z (see Figure 5). The operating system configures the virtual page table, while the hypervisor configures the EPT.

Similarly to virtual page tables, the EPT allows the hypervisor to specify the access rights of each guest physical page. In case the guest attempts to access a physical page that is either unmapped or has insufficient access rights, an event called EPT-violation occurs; triggering a VM-exit.

The Input-output Memory Management Unit (IOMMU) is an additional memory translation mechanism. In contrast to virtual page tables and EPT, IOMMU translates addresses that are accessed not by the processor but by hardware devices. Interestingly, the configuration tables of EPT and IOMMU are almost identical.

Protection from malicious access (reading and writing) to the hypervisor code and data is carried out via a special and identical configuration of the EPT and IOMMU mapping. According to this configuration, none of the sensitive memory regions are mapped, and they are therefore inaccessible from the guest or a hardware device. The EPT and the IOMMU mappings protect the memory of the hypervisor from malicious modifications.

The memory on which the hypervisor is stored is reported to the operating system as unavailable. This report is made by the UEFI firmware in case of a UEFI boot, and by the hypervisor in case of a BIOS boot (Section 2.4). A legitimate working operating system should not attempt to access this memory, and an access attempt causes an EPT violation, triggering a VM-exit.

2.5.3 Cache Evictions

The hypervisor memory protection assures that the hypervisor cannot be tampered with while in memory. Depending on the provided security service, the hypervisor may also need to store secret data (e.g., a decryption key) [RZ13]. An attacker with bus sniffing capabilities may exploit the processor's cache mechanism to extract this sensitive data.

The cache of Intel processors has three levels (see Figure 6). The first is the fastest but the smallest, and the last is the slowest and the largest. The last level cache (LLC) is divided into several *slices* of equal size, each of which has 2,048 sets, each of which, in turn, has multiple 64-byte lines (The number of lines in a set is called *associativity*). The processor evicts data from the cache to the main memory either as a response to a special instruction (e.g., `WBINVD`), which can be intercepted by the hypervisor, or to store some new data. When an instruction accesses the physical address x, the processor determines the cache set in which the data at address x should be stored: The slice is determined by applying an unknown hash function on bits 6–63 of x, while the set number is determined by bits 6–16 of x (bits 0–5 determine the bytes number inside the cache line). After the set is determined, the processor selects one of the lines in the set for eviction. We note that access to address x can cause eviction of data from address y only if x and y are mapped to the same set. Each page is mapped to $4096/64 = 64$ consecutive sets and only the $2048/64 = 32^{\text{nd}}$ page following it will potentially (since there is more than one slice) be mapped to the same sets. This observation allows the hypervisor to protect its sensitive data (e.g., a decryption key) from cache eviction attacks, by reserving the pages $0, 32, 64, 96$, and so on for its use. This technique is referred to as page colouring, and Figure 4 depicts the mapping of the hypervisor when page colouring is used. Before establishing the new mapping, the hypervisor copies the contents of pages whose index is a multiple of 32 to the pages that correspond to them in the mapping. To understand the necessity of this step, consider the following scenario. The firmware stores some value on page 32 before the hypervisor's initialisation, and loads this value from page 32 after the initialisation. Let us assume that page 32 is mapped to page 10,032, the first access will store some value to physical page 32, but the second access will load the value from page 10,032. Therefore, the contents of page 32 must be copied to page 10,032. The hypervisor uses the pages whose indices are a multiple of 32 to store sensitive information, such as a decryption key. Hence, we refer to these pages as *safe pages*.

2.5.4 Cryptographic Key

Trusted Platform Module (TPM) is a standard that defines a hardware device with non-volatile memory and a predefined set of cryptographic functions. The device itself can be implemented as a standalone device mounted on the motherboard; alternatively, it can be embedded in the CPU packaging [LPT07, p. 139]. Each TPM is equipped with a public/private key-pair that can be used to establish a secure communication channel between the CPU and the TPM. The non-volatile memory is generally used to store cryptographic keys. The processor of a TPM can decrypt data using a key stored in its memory without transmitting this key on the bus. One of the primary abilities of the TPM is environment integrity verification. The TPM contains a set of Platform Configuration Registers (PCRs) that contain (trustworthy) information about the current state of the environment. These registers can be read but cannot be assigned. The only way to modify the value of these registers is by calling a special function, `Extend(`D`)` that computes a hash of the given value D and the current value of the PCR and

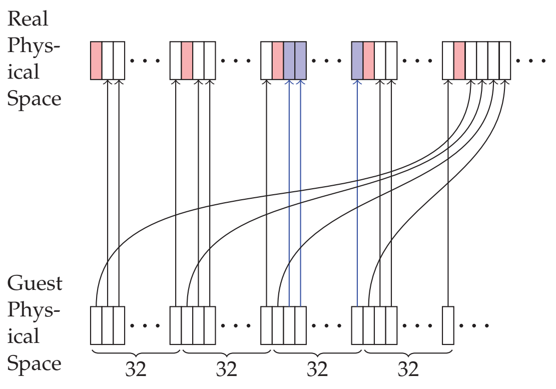


FIGURE 4 EPT configuration that maps guest physical pages (bottom) to real physical pages (top). Real physical pages whose index is a multiple of 32 (red rectangles) are safe pages, which are used exclusively by the hypervisor. Guest's physical pages whose index is a multiple of 32 are mapped to the last pages in the real physical address space. The code and the data of the hypervisor (blue rectangles) are mapped as read-only (blue lines) to the guest physical address space.

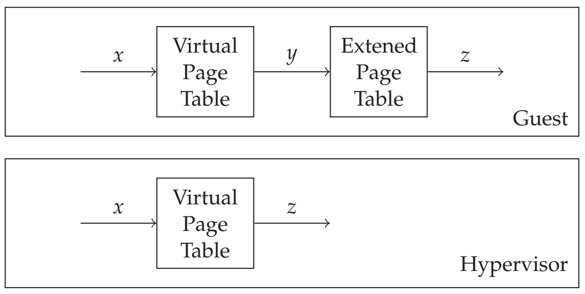


FIGURE 5 Address translations in the guest and the hypervisor. When the guest accesses a virtual address x, it is translated to a guest physical address y via the guest's virtual page table, then y is translated to a real physical address z via EPT. When the hypervisor accesses a virtual address x, it is translated to a real physical address via the hypervisor's virtual page table.

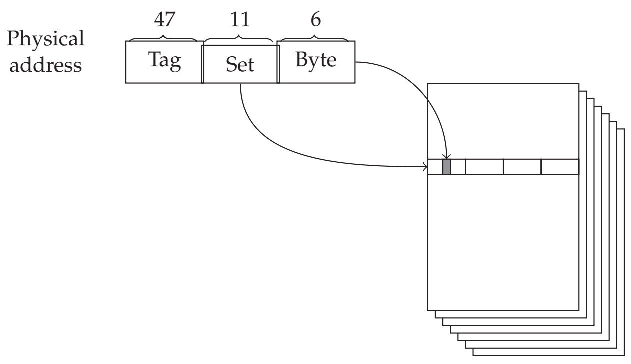


FIGURE 6 An example of the last level cache organisation of associativity four with six slices and 2,048 sets in each slice. The six least significant bits of the physical address select the byte in the cache line. The next 11 bytes select the set. The slice is determined by applying a hash function to the set and the tag fields of the physical address.

sets the result as the new value of PCR. The firmware is responsible for initialising the PCRs and for extending them with the code of a boot application before jumping to these applications. An application that wants to check its integrity can compare the relevant PCR with a known value.

Another critical ability of the TPM is symmetric cryptography. The TPM provides two functions: `SEAL` and `UNSEAL`. The first function encrypts a given plaintext and binds it to the current values of the PCRs, while the second function decrypts the given ciphertext but only if the PCR values are the same as when the `SEAL` function was called.

Algorithm 3 Boot Application Key Initialisation Sequence

```
1: if FileExists('key.bin') then
2:     EncryptedKey ← FileRead('key.bin')
3:     Key ← Unseal(EncryptedKey)
4: else
5:     Key ← Input()
6:     EncryptedKey ← Seal(Key)
7:     FileWrite('key.bin', EncryptedKey)
8: Extend(0)
```

Algorithm 3 presents the cryptographic key initialisation sequence of our boot application. The application first checks whether a file named 'key.bin' already exists (line 1). If so, its contents are read and unsealed producing a cryptographic key (lines 2–3), which is then stored in a safe page. If, however, the 'key.bin' file is missing, the application asks the user to type the key (line 5), which is then sealed and stored in a file (lines 6–7). In any case, the PCRs are extended with a (meaningless) value (line 8), thus preventing other boot applications and the operating system from unsealing the contents of the 'key.bin' file.

The `UNSEAL` function is executed by the TPM and its result is transmitted to the CPU over the bus. To protect the cryptographic key in the presence of a bus sniffer, our boot application establishes a secure communication channel (OIAP session). During the initialisation of the channel, the application encrypts the messages using the public part of the key that is embedded in the TPM.

2.6 Thin Hypervisor Transparency

Depending on the security service, the hypervisor may need to protect itself (and, possibly, its actions) from detection. An example of this is a security service that allows malware monitoring, in which case sophisticated malware could attempt to detect the presence of the hypervisor and consequently alter the behaviour of the malware. Furthermore, the malware could also attempt to detect memory anomalies in case the hypervisor modifies the memory of the guest operating system or its applications.

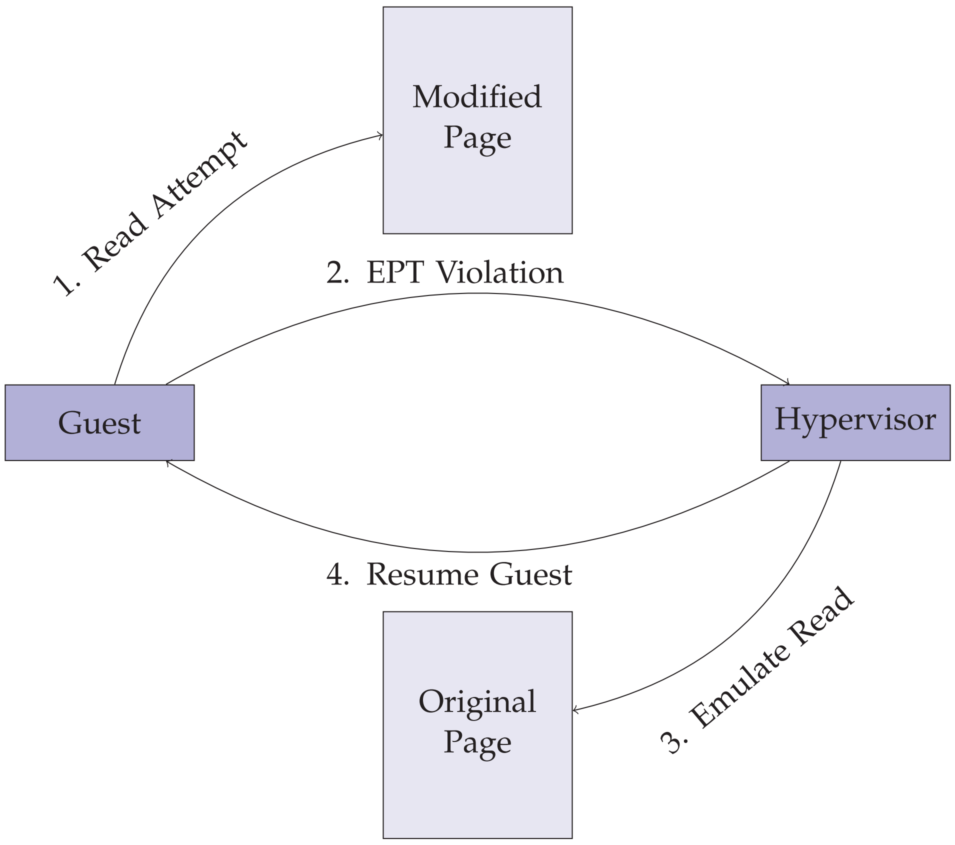


FIGURE 7 Hypervisor memory modifications transparency. First, the guest attempts to read from the modified page. Because this page is configured to execute only permissions, an EPT violation occurs. Next, the hypervisor emulates the read instruction as if it had been done on the original memory page. Finally, the hypervisor resumes the execution of the guest.

As described in Section 2.5.2, the SLAT (EPT) mechanism allows the hypervisor to also configure the access rights of the guest physical memory. Using EPT, the hypervisor can make any code modifications to the operating system and its applications completely transparent to the guest. The hypervisor configures the EPT as follows to provide memory transparency. First, the hypervisor configures the memory pages (those it desires to hide) to have execute-only permission. Then, in the case of a guest read or write attempt, an EPT violation occurs, triggering a VM-exit. Finally, the hypervisor emulates the read/write instruction as if it had been done on the original page. Figure 7 illustrates this process.

Various publications exist on the subject of hypervisor detection. For example, Garfinkel et al. discuss general virtualization anomalies [GAW+07] while Franklin et al. [FLM+07, ZR15] discuss the remote detection of hypervisors using timing attacks. Timing attacks, both local and remote, can be carried out even in the presence of a thin hypervisor. Local timing attacks can be dealt with quite easily, as local timers can be controlled by the hypervisor. Remote timing attacks (e.g., [BB05]) are more difficult to address. However, sophisticated malware cannot rely on the fact that an internet connection is available [L11]. We discuss this limitation in Section 5.2.

2.7 Thin Hypervisor Performance

A thin hypervisor generally provides a certain security service and goes into action when that service is required. However, particularly in the case of a type 1 hypervisor, it also affects overall performance of the system even when it is idle. The evaluation presented below answers the question of how the mere existence of the hypervisor degrades the system performance. To test the impact of the hypervisor on the overall system performance, we chose three benchmarking tools for Windows:

1. PCMark 10 – Basic Edition. Version Info: PCMark 10 GUI – 1.0.1457 64 SystemInfo – 5.4.642, PCMark 10 System 1.0.1457
2. PassMark Performance Test. Version Info: 9.0 (Build 1023) (64-Bit)
3. Novabench. Version Info: 4.0.3 November 2017

All the experiments were performed in the following environment:

- CPU: Intel Core i5-4570 CPU @ 3.20GHz (4 physical cores)
- RAM: 8.00 GB
- OPERATING SYSTEM: Windows 10 Pro x86-64 Version 1709 (Build 16299.248)
- C/C++ Compiler: Microsoft C/C++ Optimising Compiler Version 19.00.23026 for x86

Each tool performs several tests and displays a score for each test. We invoked each tool twice, once with and once without the hypervisor. The results of PCMark, PassMark, and Novabench are depicted in Figures 8, 9, and 10, respectively. We can see that the performance penalty of the hypervisor is, on average, approximately 5%.

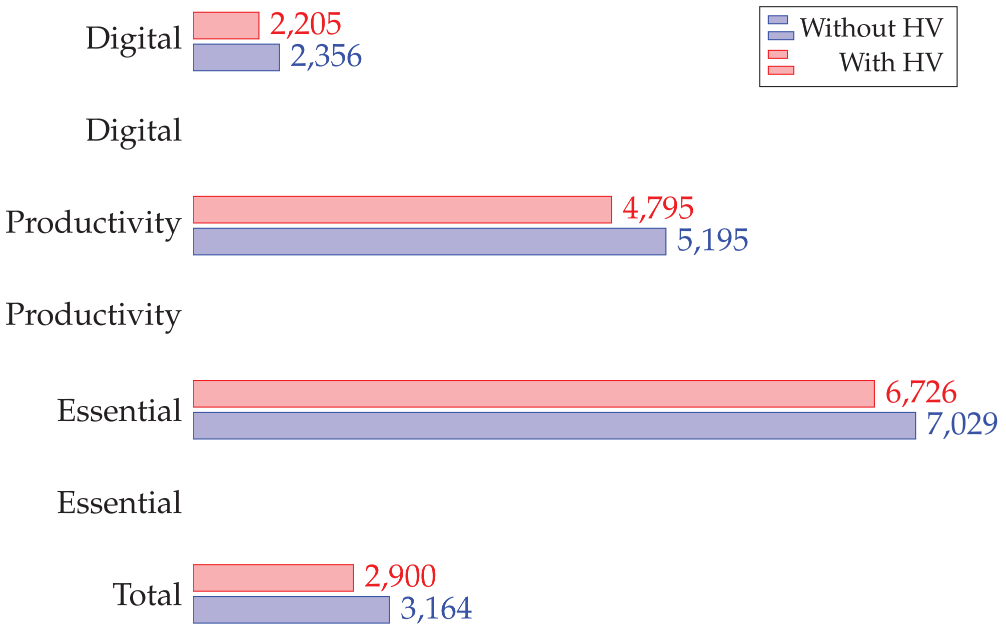


FIGURE 8 The scores (larger is better) reported by PCMark in four categories: Digital Content Creation, Productivity, Essential and Total.

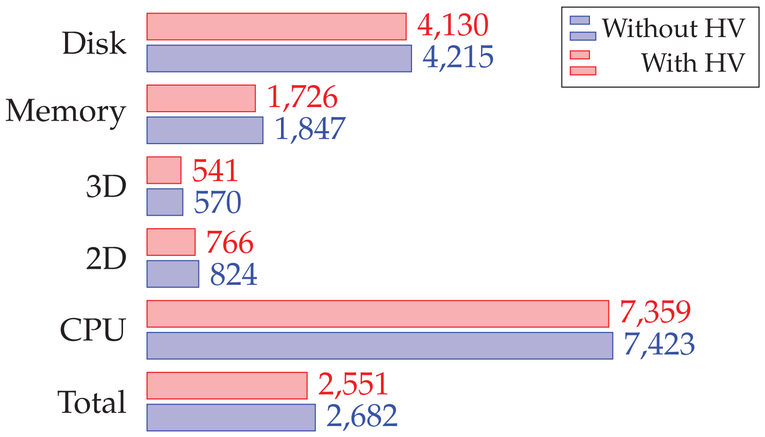


FIGURE 9 The scores (larger is better) reported by PassMark in six categories: Disk, Memory, 3D, 2D, CPU, and Total.

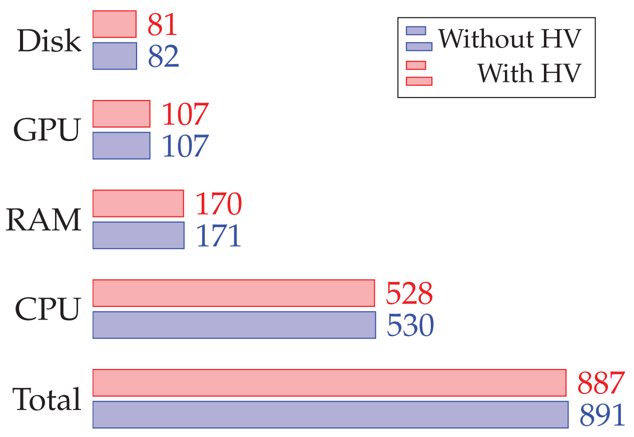


FIGURE 10 The scores (larger is better) reported by Novabench in five categories: Disk, GPU, RAM, CPU, and Total.

3 HYPERVISOR APPLICATIONS IN SECURITY

The following subsections describe in detail how the proposed architecture (Section 1.3.1) was used by the author to solve the security problems presented in Section 1.2. For each of the problems, we first review previous work done in the corresponding field and how it differs from our method. Then, we thoroughly discuss the operation of our method. Finally, we analyze the performance impact of our method.

3.1 Memory Forensics

3.1.1 Previous Work

Both software and hardware solutions have been proposed for the problem of memory acquisition. Most software solutions are susceptible to vulnerabilities in the operating system, while hardware solutions are unable to produce an atomic memory image.

Previous versions of both Windows and Linux provided a device file (e.g., */dev/mem* in Linux) that allowed the physical memory to be acquired without any special tools. However, in the event that the system is compromised, an attacker could corrupt this file [CG04], thereby rendering it useless.

Another method of memory acquisition is based on generic or dedicated hardware. Previous work has shown how a generic FireWire card can be used to acquire memory remotely [ZWZ+10]. A dedicated PCI card called Tribble works similarly [CG04]. The main advantage of a hardware solution is the ability of a PCI card to communicate with the memory controller directly, thus providing a reliable result even if the operating system itself is compromised. However, hardware solutions have three deficiencies:

1. They are expensive.
2. The memory image that they produce is not atomic.

3. These tools fail when Device Guard [DB17], a security feature introduced in Windows 10, is enabled.

In 2010 [MFP+10] and 2017 [QXM17] hypervisor-based solutions for memory acquisition have been proposed. Both allow an atomic memory image to be obtained; these methods work as follows:

1. When the hypervisor is requested to start memory acquisition, it configures all memory pages to be non-writable.

2. When an attempt is made to write to a memory page *P*, the hypervisor is notified.

3. The hypervisor copies the contents of *P* to its internal buffer and configures the page *P* to be writable.

4. The hypervisor periodically sends the data in its internal buffer (e.g., via a network). If more data can be sent than is available in the internal buffer, then the hypervisor sends other pages and configures them to be writable.

Both of these methods, though recent, have been tested on Windows XP SP3 with a single enabled core. Furthermore, these methods propose to configure all memory pages to be non-writable without exception. However, we observed in modern operating systems that locking all of the memory pages is not possible, as the access time to some of the pages cannot be delayed. Consider the following example for such pages. Generally, interrupt service routines react to interrupts in two steps: They register the occurrence of an interrupt and acknowledge the device that the interrupt was serviced. The acknowledgement must be received on time; therefore, the registration of an interrupt, which involves writing to a memory page, must not be intercepted by a hypervisor, that is, these pages must remain writable.

Our method works similarly to [MFP+10] and [QXM17]. However, our method works in modern operating systems that support multicore architecture.

3.1.2 Operation

The hypervisor remains idle (or deactivated) until it receives a memory acquisition request. When the request is received, the hypervisor performs two actions:

1. Locates and copies the page to which write access must not be blocked.

2. Requests all processors to configure the access rights of all memory pages to be non-writable.

Each processor configures the EPT such that all memory pages are non-writable. An attempt to write to a page *P* will trigger a VM-exit, thus allowing the hypervisor to react. The hypervisor reacts by copying *P* to an internal queue and making *P* writable again. Future attempts to write to *P* will not trigger a VM-exit.

The queued pages are transmitted to a remote machine via a communication channel, which may or may not be secure depending on the security assumptions about the underlying environment. The size of the queue is dictated by the communication channel bandwidth and the volume of pages that are modified by the system. If the communication channel allows more data to be sent than is available in the queue, then the hypervisor sends other non-writable pages and configures them to be writable. This process continues until all pages become writable.

To locate the pages to which write access cannot be blocked, the hypervisor must be familiar with the specific operating system internals. These memory regions can be a part of the operating system static executable (code or data) or dynamic regions. The location of these regions is even more challenging to obtain in operating systems which employ Address Space Layout Randomisation (ASLR). When ASLR is enabled, the operating system splits its virtual address space into regions. Then, during the initialisation of the operating system, each region is assigned a random virtual address. These regions (their ranges) must be stored by the operating system. To find these regions, and due to ASLR, the hypervisor must compute the location in which these regions are stored relative to a known base. We chose the system call service routine, whose address is stored in the LSTAR register, as the known base. We show how these memory regions can be obtained in Windows 10 (x64) in Paper [PII].

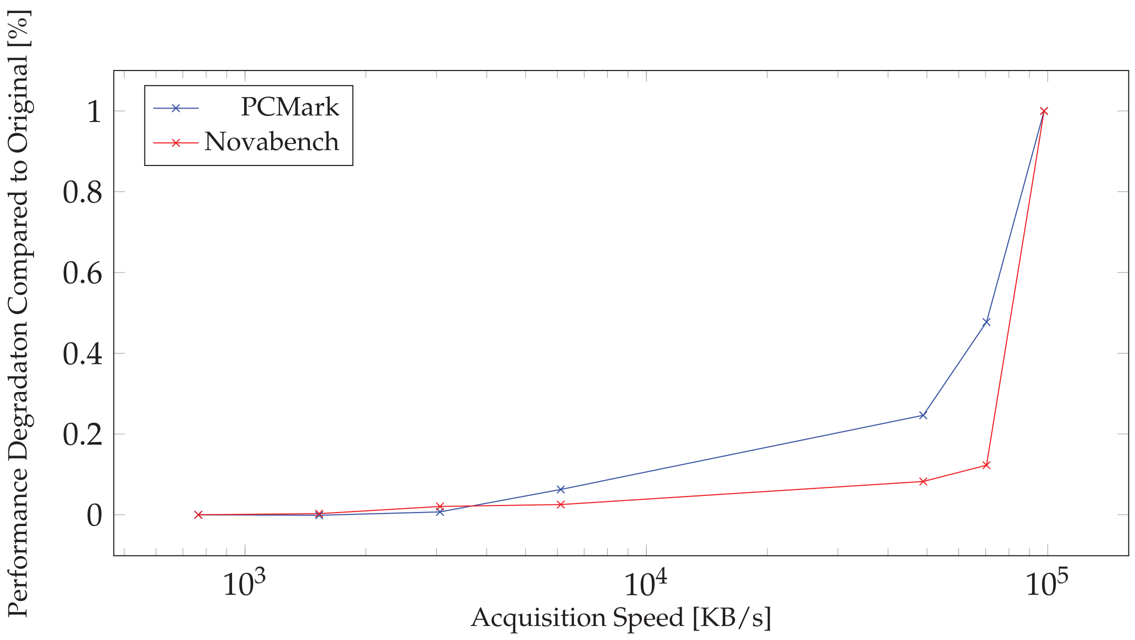


FIGURE 11 Performance degradation due to memory acquisition.

3.1.3 Analysis and Performance

The evaluation of our method can be divided into two parts: (1) memory usage and (2) memory acquisition performance. The memory used by the hypervisor can be divided into three main parts:

1. The code and the data structures of the hypervisor.
2. The EPT tables used to configure the access rights to the memory pages.
3. The queue used to accumulate the modified pages.

The size of the queue is mainly dictated by the number of the pages to which write-access cannot be blocked. In Windows 10 (x64), we have shown that the required size is approximately 60MB.

To evaluate the memory acquisition performance, we studied the correlation between the speed of memory acquisition and the overall system performance. If the system is responsive during the acquisition then it is considered successful. Figure 11 depicts the results. The horizontal axis represents the memory acquisition speed. The maximal speed we could achieve was 97920KB/s, at which speed the system became unresponsive and the benchmarking tools failed. The vertical axis represents the performance degradation (in percent) measured by PCMark and Novabench. More precisely, $t_i(x)$ denotes the total result of benchmark $i = 1, 2$ (for PCMark and Novabench, respectively) with acquisition speed of x; the performance degradation $d_i(x)$ is given by $d_i(x) = 1 - \frac{t_i(x)}{t_i(0)}$.

3.2 Malware Analysis

3.2.1 Previous Work

Current dynamic malware analysis methods can be classified into four categories: (1) hooking methods, (2) emulation methods, (3) hypervisor-based methods, and (4) bare-metal based methods. Hooking methods [WHF07, GF10, YMH+17] perform in-line code overwriting of API code directly in the process memory. Therefore, the malware attempts to use any of the Windows APIs can be monitored, specifically, by a special module (typically, a DLL) that is injected into the monitored process address space. Burguera et al. [BZN11] used the *strace* tool available in Linux for the purpose of malware analysis. However, this method can be easily detected by a sophisticated malware [CAM+08]. Though these methods are very efficient, they also have several significant deficiencies. These methods are:

1. detectable — since these methods directly modify the process memory, malware can simply check the contents of the desired API function against a known signature to check whether it has changed. For example, the structure of the Windows Native API (i.e., *ntdll.dll*) stubs can be easily determined and have a similar structure in different Windows versions. In addition, the monitoring module may be detected by asking the operating system for a list of loaded modules (see *skippable*).

2. modifiable — malware can simply remove or replace the hooking code by writing directly into the process memory. If *DEP* is in use, the malware may change the access permissions of the code (e.g., using *VirtualProtect* in Windows) to get write permissions (see *skippable*).

3. skippable — malware can skip operating system API calls by performing system calls directly. Even though the system call numbers vary between different version of an operating system, we claim that this is completely feasible, as it is sufficient to detect the exact operating system version (e.g., using the *NtQueryValueKey* native API function in Windows).

Emulation methods, e.g., [BKK06], perform system call tracing without inducing any modifications to the running guest. However, these methods are slow and may be exploited due to incomplete emulation [F06, F07, RKK07, DRS+08, VC14].

Hypervisor-based methods [DRS+08, LMP+14] provide a malware analysis system that is both transparent (i.e., performs no modifications to the running operating system) and is more efficient compared to emulation methods. Both [DRS+08] and [LMP+14] are built on top of the XEN hypervisor [BDF+03] and consequently run the malware sample inside a regular domain, which remains untouched, while the actual analysis takes place in the control domain (dom0). The latter modus operandi forces the following steps upon an event in the regular domain (i.e., the domain on which the malware runs):

- A transition from the regular domain to the XEN hypervisor.
- A transition from the XEN hypervisor to the control domain.
- A transition from the control domain to the XEN hypervisor.
- A transition from the XEN hypervisor to the regular domain.

As a consequence, these methods incur a significant overhead and are also vulnerable to attacks on the XEN hypervisor [DWH+12]. Moreover, the mere existance of the XEN hypervisor also incurs a non-negligible overhead [LWJ+13]. All hypervisor-based methods are vulnerable to VMM detection attacks [FLM+07, GAW+07, BBN12].

Bare-metal based methods [KVK11, KVK14] provide a bare-metal system for malware monitoring and are therefore not prone to hypervisor detection attacks. However, the extraction of a behavioural profile of malware from such systems is difficult because an analysis component must be installed on the target machine. The presence of such a component can be detected by sophisticated malware. Therefore, these methods are best suited for tracking activities that can be externally monitored (e.g., network and disk activities).

Our method can be classified as a hypervisor-based method. However, in our method:

1. The system call monitoring component is located within the guest operating system and consequently no intervention of the hypervisor is required during the system call monitoring and analysis. As a result, the performance overhead is kept to a minimum.

2. Though completely handled in the guest context, the monitoring component is fully transparent to the guest operating system.

The code of the hypervisor consists of 6,000 lines of code and the code of the monitoring component consists of 2,000 lines of code. This hypervisor's code size is significantly smaller than that of other hypervisors [MMH08] [VMware]. Moreover, our hypervisor provides no direct API calls (e.g., through the `vmcall` instruction) to the guest operating system. We believe that the latter points improve the reliability of our hypervisor.

3.2.2 Preparations

We refer to the entity wishing to analyse malware as the *malware analyst*.

A system call typically receives a list of parameters that describe the request of the caller. For example, *NtOpenFile* receives six arguments, of which four are input arguments, which describe the properties of the file to be opened (e.g., its path, desired access, etc.). The length of the argument list is limited. This limit is different for each operating system. For example, in Windows 7 x86, the maximum number of arguments a system call may receive is 17. A trivial system call analyser may use this fact to copy the first 17 arguments of each system call. A special analyser can later be used to filter the unnecessary arguments of each system call. However, this method of system call analysis is ineffective, as a system call parameter can point to a hierarchy of meaningful information. For example, the third parameter of *NtOpenFile* is a pointer to a *POBJECT_ATTRIBUTES* structure, which, in turn, has a field that points to a *PUNICODE_STRING* structure, which, in turn, has a field that points to a buffer that contains the actual path of the file to be opened; see Figure 12. Figure 13 presents a simple sandbox configuration file (valid for Windows 7 x86) containing a single system call entry. Line 1 provides information regarding the specific system call. Specifically, its number and the number of relevant parameters. Lines 2, 3, 8, and 9 provide information regarding the system call parameters. Each parameter line is composed of the parameter index (0 for the first parameter, 1 for the second, and so on), its type, and type-specific data. For example, line 2 describes the second parameter of *NtOpenFile*, namely, *DesiredAccess*. The parameter is a primitive type, with a size of 4 bytes. Lines 4 and 6 describe a pointee of a structure type. Each structure type is composed of one or more fields; for example, line 4 describes a structure of type *OBJECT_ATTRIBUTES*, which is pointed by the third parameter (*ObjectAttributes*). The structure is composed of a single pointer field at an offset of 8 bytes. This field points to another structure field of type *UNICODE_STRING* which is composed of a single field located at an offset of 4 bytes and is of buffer type.

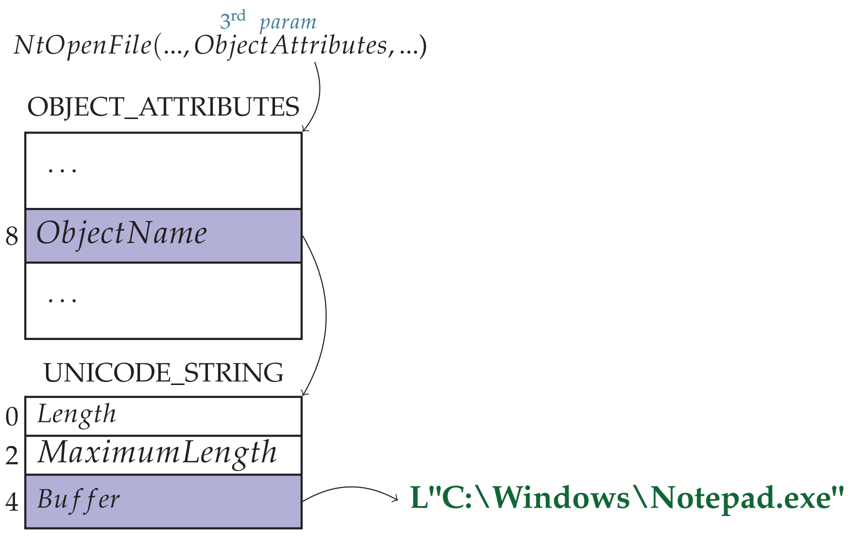


FIGURE 12 NtOpenFile 3rd parameter information hierarchy in a x86 Windows system. The *ObjectAttributes* parameter points to a *OBJECT_ATTRIBUTES* structure. A field named *ObjectName* is located at a offset of 8 bytes in the *OBJECT_ATTRIBUTES* structure. This field points to a *UNICODE_STRING* structure. A field named *Buffer* is located at a offset of 4 bytes in the *UNICODE_STRING* structure. This field points to the unicode string "C:\Windows\Notepad.exe".

Finally, the malware analyst must install the boot application and the configuration file (see Section 2.4).

```
s,179,4 ; NtOpenFile
p,1,pri,4,ep ; (ACCESS_MASK) DesiredAccess
p,2,ptr,p ; (POBJECT_ATTRIBUTES) ObjectAttributes
        p,struct,f ; (OBJECT_ATTRIBUTES)
                f,8,ptr,p ; (PUNICODE_STRING) ObjectName
                p,struct,f ; (UNICODE_STRING)
                        f,4,buffer,0,2,ef ; (PWSTR) Buffer
p,4,pri,4,ep ; (ULONG) ShareAccess
p,5,pri,4,ep ; (ULONG) OpenOptions
```

FIGURE 13 Malware Analysis – Sandbox Configuration File Example for Windows 7 x86

3.2.3 Operation

The performance of system call interception can be improved by performing the interception in the context of the guest operating system (i.e., no transitions to the hypervisor are required). To achieve the latter, the hypervisor must map the monitoring component into the address space of the guest operating system.

We developed a monitoring component, which is bascially a kernel application that does not depend on external libraries. Nevertheless, it can use functions provided by the kernel itself (for example, the *Zw* family in Windows). The monitoring component supports the Microsoft ABI but can be easily ported to support other ABIs (e.g., System V).

A possible approach for mapping the monitoring component into the address space of the guest operating system is by directly modifying the operating system page tables (specifically, the upper kernel part). However, this method is intrusive and may cause system instabilities. For example, in Windows 10, we observed that the Memory Management Unit of Windows generates a BSOD on such attempts. An alternative method is to virtualize the page-tables of the guest operating system (a technique known as shadow-page-tables). However, this method is both complex and may be inefficient.

Because of these deficiencies, we decided to map the monitoring component into the address space of the guest operating system as follows. The hypervisor waits for the operating system kernel to be successfully loaded. During its initialisation, the operating system kernel writes to the system call handler register. The hypervisor uses this event as an indication for the operating system load. The hypervisor configures the guest such that writes to the system call handler register cause a VM-exit. When such an attempt occurs, the hypervisor looks for the kernel image within the guest memory and uses its export table to find the address of *ExAllocatePool*. Figure 14 depicts the guest memory layout upon a VM-exit caused by writing to the system call handler register. The hypervisor then configures the guest to cause a VM-exit upon a breakpoint exception. Next, the hypervisor replaces the first byte of the *ExAllocatePool* function with a breakpoint instruction (`INT3` in x86) and configures the guest such that a VM-exit is triggered upon a breakpoint exception. Algorithm 4 depicts the actions of the hypervisor upon the next call to *ExAllocatePool*. The hypervisor differentiates between two cases:

1. Entry to *ExAllocatePool* – First, the hypervisor saves the return address and the original arguments (lines 3-4). Then, the hypervisor replaces the original arguments with arguments of its own (e.g., the amount of memory required for the monitoring component) (line 5). Next, the hypervisor restores the original entry byte and saves the original return byte (lines 6-7). Finally, the hypervisor sets a breakpoint trap in the return address (line 8).

2. Return from *ExAllocatePool* – First, the hypervisor copies the monitoring component to the memory returned by *ExAllocatePool* (line 10). Then, the hypervisor restores the original arguments of the function (saved in the entry case) (line 11). Finally, the hypervisor restores the original exit byte and sets the instruction pointer to the entry of *ExAllocatePool* (lines 11-12). This step effectively re-executes the function with its original arguments.

After the allocation completes, the hypervisor forfeits the control over the system call handler register.

Finally, in order to intercept every system call conducted by the guest OS, the hypervisor hooks the original system call handler such that it calls the mon-

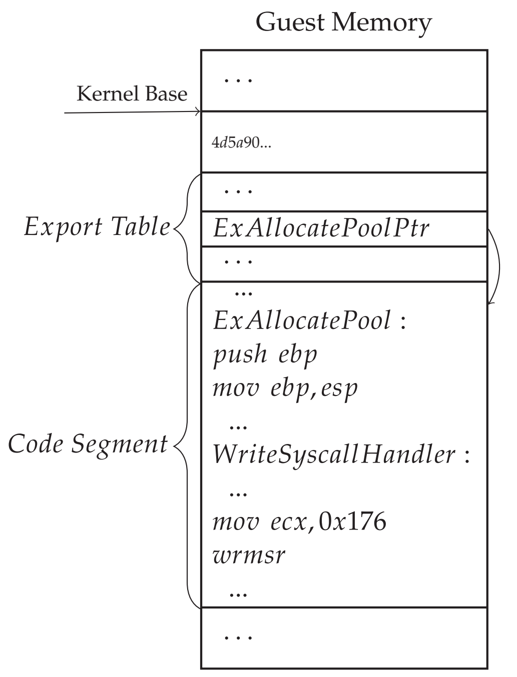


FIGURE 14 Windows memory layout after performing a write to SYSENTER_EIP msr. The hypervisor intercepts the `wrmsr` instruction. Upon interception, the hypervisor looks for the kernel-image base by moving backwards at a page granularity until the PE magic number (0x4d5a90) is encountered. Finally, the hypervisor looks for the *ExAllocatePool* address within the export table.

itoring component (which is now mapped into the address space of the guest OS).

Algorithm 4 Malware Analysis – Hypervisor Breakpoint Handler

```
 1: procedure HANDLEBREAKPOINT()
 2:     if IP is in ExAllocatePool entry then
 3:         Save return address
 4:         Save original args
 5:         Modify original args
 6:         Restore original entry byte
 7:         Save original return byte
 8:         Set return byte to 0xCC
 9:     else if IP is in return address then
10:         CopyGuestApplicationToMemory()
11:         Restore original args
12:         Restore original exit byte
13:         Set IP to entry of ExAllocatePool
```

The monitoring component intercepts all system calls and records only those belonging to the monitored process. Initially, the monitoring component sets up a sandbox by parsing the configuration file. A system call conducted by a monitored process is analysed and then recorded, and the recorded data is written into an internal memory buffer and may be written to an external source every time the buffer becomes full so it can be reused. The external source can be a remote server, a local file, or a large, previously allocated memory buffer. At the end of each interception, our handler jumps to the original operating system handler. Algorithm 5 depicts the monitoring component system call handler. First, the system call handler fetches the current process, the system call arguments, and the current monitored process (lines 2-4). The system call handler then distinguishes three cases:

- Current process is a monitored process and a monitored process is not active — in this case, the system call handler initializes a new monitored process (lines 5-7).
- A monitored process is active and has initiated an exit system call (e.g., *NtTerminateProcess* in Windows) — in this case, the system call handler dumps the current recorded data to an external source and frees the monitored process resources (lines 8-9).
- A monitored process is active and has conducted a system call — in that case, the system call handler analyzes and records the system call. The data for analysis is fetched from the arguments according to the configuration file, provided by the malware analyst, and is sequentially written into a buffer of fixed length. If a configuration is not provided for a system call, the maximum possible number of arguments (e.g., 17 in Windows 7 x86) are fetched. When the buffer becomes full, the data is written to an external source so it can be reused (line 10-11).

Finally, the system call handler jumps to the original operating system handler (line 12).

Algorithm 5 Monitoring Component – System Call Handler

```
1:  procedure HANDLESYSTEMCALL($scNo$)
2:      $args \leftarrow GetArgs()$
3:      $cp \leftarrow GetCurrentProcess()$
4:      $mp \leftarrow GetMonitoredProcess()$
5:      if $NotActive(mp)$ and not Exit(scNo) then
6:          if $IsMonitoredProcess(cp)$ then
7:              $MonitorProcessInit(mp)$
8:      else if $Active(mp)$ and Exit(scNo) then
9:          $MonitoredProcessDumpAndFree(mp)$
10:     if $Active(mp)$ and IsMonitored(cp) then
11:         $MonitoredProcessAnalyse(mp, scNo, args)$
12:     $JumpToOriginalHandler()$
```

3.2.4 Analysis and Performance

In this section, we discuss the overall performance impact of our system. Specifically, we determine how the overall performance is affected by (1) the hypervisor, (2) the monitoring component, and (3) analysing a process. In Paper [PVI], we also discuss the process behaviour analyser and show an analysis of a sample malware. All the experiments were performed in the following environment:

- CPU: Intel i7-7500 CPU @ 2.70GHz
- RAM: 16GB
- OPERATING SYSTEM: Windows 7 SP1 x86

In this experiment, we tested the system in three scenarios:

- without hypervisor.
- with hypervisor and disabled monitoring component.
- with hypervisor and enabled monitoring component.

We picked three benchmarking tools for Windows:

(a) PCMark 10 – Basic Edition. Version Info: PCMark 10 GUI – 1.0.1457, SystemInfo – 5.4.642, PCMark 10 System 1.0.1457.

(b) PassMark Performance Test. Version Info: 9.0 (Build 1023) (32-Bit).

(c) Novabench. Version Info: 4.0.3 November 2017.

Each tool performs several tests and displays a score for each test.

As can be seen in the results reported in Figure 15, the hypervisor degrades the performance by no more than 5%, and the monitoring component degrades the performance by, at most, an additional 1%.

Furthermore, we checked the performance overhead of a monitored process. We picked an application that performs heavy usage of system services, the PassMark benchmark tool (the one described in the previous paragraph). Recall that the recorded system calls are first stored in a memory buffer. When the buffer becomes full, it is written to an external source. We used a buffer size of 122,000 bytes and a regular file on the disk as the external source. The final dump file was 44MB in length. We have not observed any noticeable difference in performance.

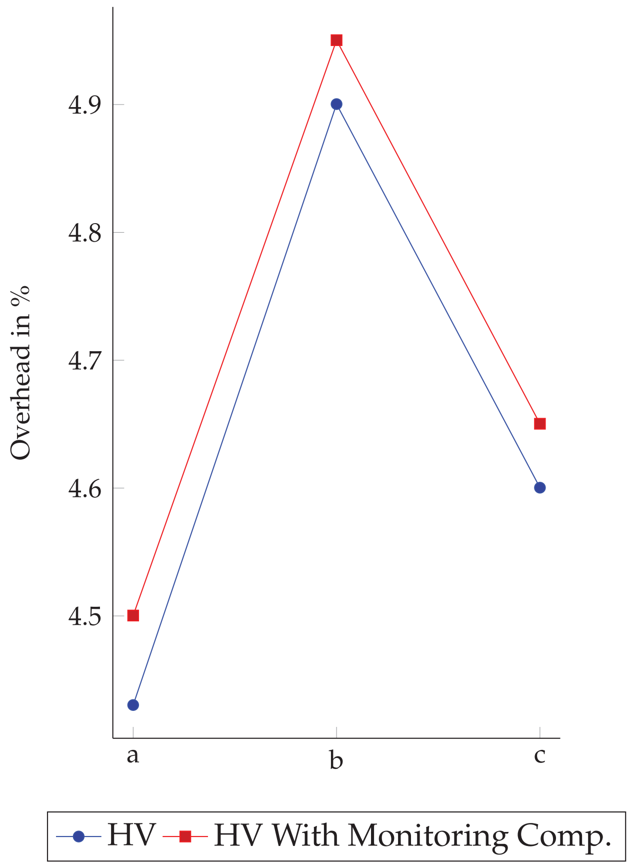


FIGURE 15 Overhead of the benchmark execution under two conditions: (a) with HV only, and (b) with HV and enabled monitoring component

3.3 Unauthorised Execution Prevention

3.3.1 Previous Work

Current methods of preventing unauthorised execution can be divided into three categories [IM07]:

1. Behavioural: These systems analyse the behaviour (e.g., network or hard-drive activity) of the system and compare current behaviour with a predefined behaviour. If these behaviours differ, the environment is considered breached.

2. Signature-oriented: These systems contain a database of code samples that are known to be malicious. Every loaded executable is scanned and if it contains a code sample that is present in the database, then the environment is considered breached. Most anti-virus programs can be categorised as signature-oriented.

3. Whitelist-oriented: These systems contain a database of allowed executables. The criteria used for whitelisting are frequently based on one or more

file attributes (e.g., file-path or cryptographic hash) [PRE12]. Unlike signature-based systems, only these executables are allowed to run. These systems typically intercept every loaded executable and check whether it is contained within the database. If not, then the environment is considered breached.

The strength of behavioural systems is difficult to evaluate because these systems are based on heuristics [TKM02]. Signature-based systems can protect only against attacks that were previously discovered and analysed and are, therefore, ineffective against zero-day attacks. Whitelist-based systems provide the strongest protection guarantees [M09] but are also the most restrictive. For example, to install a new program, the system administrator must allow this program to be installed by inserting it into the whitelist database. Typically, whitelist enforcement is performed by intercepting executable images at their load time (e.g., by intercepting system calls) [F17]. In the event of a vulnerability, exploitation becomes possible in runtime [B18]. Nonetheless, in environments which tend not to change frequently, the preferred option is a whitelist-based system.

Our method can be categorised as whitelist-based as it permits the creation of a whitelist database of allowed executables which is used by the system to enforce authorised execution. However, our method does not suffer from the two main deficiencies of current methods:

1. In our method, the execution of a given executable image (both in user mode and kernel mode) is enforced during its entire lifetime.

2. In our method, the system can prevent the execution of unauthorised code even in the event that an attacker has full control over the operating system kernel.

3.3.2 Preparations

We refer to the entity wishing to protect the system as the *system administrator*.

The system administrator must first scan a system that is initially trusted. We refer to the scanner tool as the *executable scanner*. The executable scanner runs on an initially trusted system. It recursively looks for all executable images, specifically, executables and shared-objects. In x86–64, memory accesses are RIP-relative: That is, the access offset to local symbols can be computed in advance by the static linker. Therefore, modifications to code that reference local symbols are not needed in runtime. Modifications to an executable image in runtime are indeed possible (for example, in Windows, relocations) by a module named the dynamic linker (or loader). These, however, can be computed in advance by the executable scanner, as they are located in the executable itself. For our prototype, we picked Linux as the operating system in which the dynamic linker performs modifications to the data segment only. Therefore, to generate the whitelist-database, the executable scanner hashes the entire code segment (in a page granularity) and stores the results consecutively (and sorted) in the whitelist-database. Figure 16 depicts this process. The kernel image (e.g., *vmlinux* in Linux) cannot be necessarily scanned in a page granularity, as the kernel performs heavy modifications

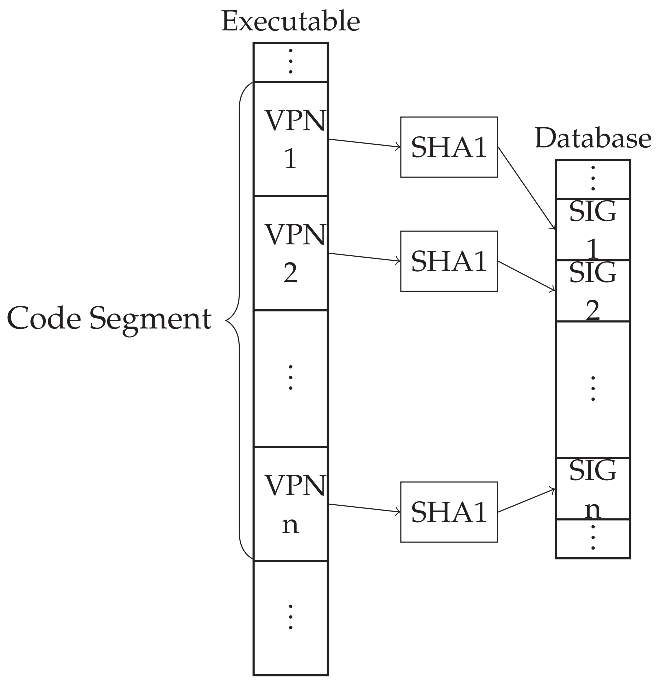


FIGURE 16 The user mode executable image to be signed (left) is composed of one code segment divided into virtual pages. Each page is signed using SHA1, and the result is stored in the whitelist database (right).

on its code upon load. However, most of the modifications, once performed, do not change anymore. Therefore, to keep the database simple, the executable scanner scans the kernel image in runtime and stores the results consecutively (and sorted) in the whitelist-database.

Next, the system administrator must install the boot application (see Section 2.4). The system administrator must store the *whitelist-database* file, which was produced by the executable scanner, on a local drive that is accessible by the firmware. During its first execution, the boot application verifies the authenticity of the whitelist database using our built-in hardcoded certificate. To ensure that the hardcoded certificate is left untouched, a secure boot method must be used (see Section 2.5.1).

3.3.3 Operation

The hypervisor must be notified of every execution attempt, and therefore, the hypervisor sets the access rights of the physical address space (except the memory regions on which it resides; see Section 2.5.2) to write-only. This step ensures that any execution attempt triggers a VM-exit, thus allowing the hypervisor to validate the faulting page.

The hypervisor waits for an EPT violation to occur, at which point the processor saves the guest's state to VMCS, loads the hypervisor's state from VMCS, and begins to execute the hypervisor's predefined VM-exit handler. The handler checks whether the VM-exit was due to an EPT violation. Among the information stored in VMCS is the EPT violation reason and the guest physical address that caused the EPT violation. Due to the nature of our method, a page can either be executable or writable, but not both. Therefore, all the EPT violations are attributed to an attempt to either write or execute.

- If the violation was due to a write attempt, the hypervisor then removes the execute rights from the violating page, grants it write rights, and performs a VM-entry.

- If the violation was due to an execution attempt, the hypervisor then computes the hash of the violating physical page and searches for the resultant hash in its whitelist database. If a match is found, the hypervisor then removes the write rights from the violating page, grants it execute rights, and performs a VM-entry. If a match is not found, then in the event that the violation occurred in user mode, the hypervisor injects a general-protection fault to the guest operating system. Otherwise, if the violation occurred in kernel mode, the hypervisor then freezes the system. Typically, the operating system reacts to general protection exception in user mode by stopping the running process.

In kernel mode, enforcing an unauthorised execution cannot always be done lazily (i.e., only at the time of a violation). In kernel mode, some actions are time-sensitive. For example, acknowledging an interrupt to the PIC cannot cause an EPT violation, as interrupt requests of equal or lower priority are not generated until the page is given execution rights and an acknowledgement is sent to the PIC. Therefore, the hypervisor verifies the kernel code pages and grants these pages execution rights in advance.

3.3.4 Analysis and Performance

Our method becomes active when an EPT violation occurs, either due to write or execute attempts. Whenever a page requires execution rights, the hypervisor computes its hash and searches for a match within the whitelist database. When a page requests write rights, the hypervisor simply removes its execution rights and grants it write rights instead. To estimate the induced overhead of this process, we took a large executable file (10MB) and modified one of its code pages such that the first byte of the page was 0xc3 (return-from-procedure opcode in x86). We wrote an application that requests a mapping of the aforementioned file into its virtual address space with full access rights (read, write, and execute). Next, using the *fadvise64* system call, we instructed the operating system not to keep the file in memory. The size of the file, along with the advice caused the access to any page within the mapped file to always generate a major page fault (i.e., it forced the operating system to access the disk). Then, using `RDTSC`, we measured the number of cycles it takes to perform the call to our modified page and return, with and without active page verification. We ran the application a total of 100,000 times. As can be seen by the results presented in Figure 17, the performance penalty of the active page verification is less than 1%.

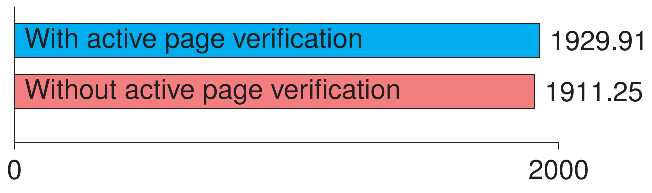


FIGURE 17 Thousands of cycles (fewer is better) for a single call and return.

To evaluate the overall performance impact of our method, we used the Phoronix Test Suite [LT18] benchmark software. We chose a subset of six tests and tested our method in three scenarios: (1) without a hypervisor, (2) with a hypervisor and without page verification, and (3) with both a hypervisor and page verification. The conducted tests were:

- unpack-linux: Linux kernel unpacking, disk-intensive, default configuration
- compress-7zip: 7-Zip compression test, cpu-intensive, default configuration
- *dbench-6client*: Dbench disk performance test, disk-intensive, 6-client configuration
- *dbench-48client*: Dbench disk performance test, disk-intensive, 48-client configuration
- *ramspeed*: System memory performance test, memory-intensive, copy and integer configuration
- *git*: Sample git operations, general system benchmark, default configuration

Figure 18 presents the results. The performance penalty of the hypervisor is no more than 5% compared with no-HV, whereas compared with the performance penalty of the HV with page verification, it is no more than 2% compared with HV only.

The proposed system suffers from two main limitations, namely, managed programs protection and native code-reuse attacks.

The first limitation derives from the fact that a managed program is not directly executed by the processor but instead by a native program. Our method provides protection for the native program but not to the programs it executes (as it is only data). A possible solution to address managed and interpreted code is to store the hashes of the managed and interpreted code within the whitelist database. The hypervisor can intercept the virtual machine application attempts to load the code and perform a hash validation. The hypervisor can detect such attempts by intercepting system calls within the guest operating system. This solution must be further extended to support Just In Time (JIT) code.

The second limitation lies in the fact that our method guarantees that only signed code is allowed to be executed. However, code-reuse attacks use known existing code to carry out an attack. These kinds of attacks are heavily researched

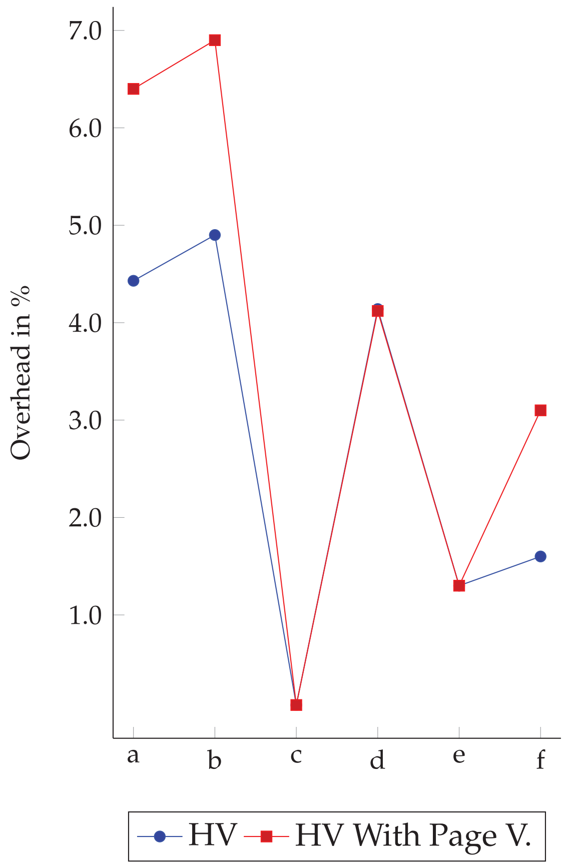


FIGURE 18 Overhead of the benchmark execution under two conditions: (a) with HV only, and (b) with HV and enabled page verification

and many solutions, which can be integrated into our solution to provide more total protection.

We refer to these two limitations in Chapter 5.

3.4 Preventing Reverse Engineering of Software

3.4.1 Previous Work

A wide range of approaches exist to code protection. Among the different methods, from an operational point of view, our system is most closely related to *encoding*-based methods. However, the security guarantees of the system are comparable to those given by *obfuscation*-based methods.

Encoding-based methods encrypt portions of a program, and then decrypt them before their execution. However, the decryption key is embedded in the decryption algorithms and can therefore be extracted. Moreover, since the code must eventually be decrypted, it can be extracted during run-time. A notable advantage of these methods is their performance. Since the encrypted code is decrypted only once, during the program's loading, the overhead of this protected method is minimal. Examples of such methods are provided in Paper [PIII]. However, these tools are vulnerable to automatic code extraction [R17].

The security guarantees of obfuscation-based methods can be summarised as follows: 'increase the reverse-engineering costs in a sufficiently discouraging manner for an adversary' [JRW+15, p. 1]. This guarantee is achieved by applying various transformations to the program that should be protected. There are numerous transformations ranging from basic instruction substitutions, which replace $a = b + c$ with $a = b - (-c)$, to control-flow flattening, which reorganises the flow between basic blocks of a function. Applying several transformations together improves the security but degrades the performance of the system. Examples of obfuscation-based methods and their resultant performance penalties are provided in Paper [PIII].

Our method can be classified as encoding since it encrypts a set of functions and then decrypts them before their execution. However, there are two advantages to our method over current methods:

1. In our method, the decryption key is not embedded in the decryption algorithm, but instead, the key is stored in a widely available hardware device (TPM).

2. In our method, the decrypted code is protected by a hypervisor during its execution, thus making the method safe even in the presence of a run-time analysis tool.

To the best of our knowledge, hypervisors have never been used for code protection.

3.4.2 Preparations

We refer to the entity wishing to protect the program as the *distributor*.

The distributor must first encrypt its sensitive code. The encryption process is performed in the granularity of a function. It receives as input a text file that specifies the executable files and the functions to be encrypted as well as the encryption key to be used. The encryption tool produces an output file that contains the encrypted versions of all the functions that were selected for encryption. We call this output file a *database*. In addition to the generation of the *database* file, the encryption tool 'erases' the instructions of the selected functions. The erasing of code is performed by replacing the original instructions by a special instruction, and we refer to the resulting file a *protected executable*. The special instruction we have selected is the `HLT` instruction. When executed in kernel mode, the `HLT` instruction causes the processor to halt. In user mode, however, this instruction generates a general protection exception, which can be intercepted by the hypervisor. Any instruction that generates an exception can qualify as a special instruction in this sense. For example, the `INT3` instruction generates a breakpoint exception and can therefore be used to replace the original instructions of the function.

Next, the distributor must install the boot application (see Section 2.4). During its first execution, the application asks the user to enter the code decryption

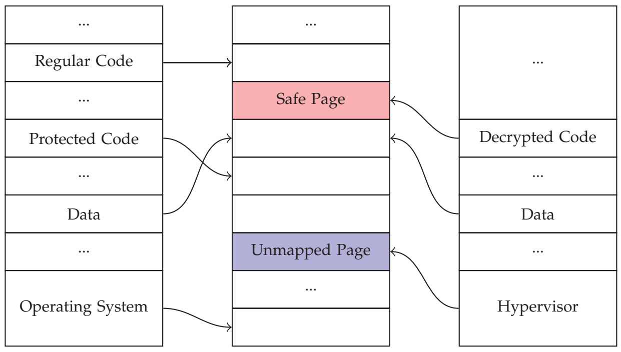


FIGURE 19 Virtual address space layouts of the hypervisor and the guest during protected function execution. The code and the data structures of the hypervisor are not mapped in the guest. The protected code is decrypted to a safe page. The virtual address of the protected function in the hypervisor corresponds to its virtual address in the guest. The mapping of the pages which store data are identical in the hypervisor and the guest. The code of the operating system is not mapped in the hypervisor.

key, after which the application encrypts the key using a TPM, a process known as *sealing*, and stores the resulting encrypted key in a local file. During subsequent executions, the application reads the file and decrypts its contents using the TPM, thus obtaining the code decryption key. It is impossible to obtain the code decryption key from another boot application (see Section 2.5.4). The distributor should store the *database* file, which was produced during the encryption phase, on a local drive that is accessible by the firmware.

3.4.3 Operation

A *protected executable* can run as usual without any interference when only functions that were not selected for encryption are called, and meanwhile, the hypervisor silently waits for an encrypted function to be called. Recall that the encryption tool replaces the original instructions of a function which is selected for encryption by a special instruction which generates an exception. The hypervisor is configured to intercept that specific kind of exception, namely the general protection exceptions. When such an exception is generated, a VM-exit occurs. The processor saves the guest's state to VMCS, loads the hypervisor's state from VMCS, and begins to execute the hypervisor's defined VM-exit handler. The handler checks whether the general protection exception was caused by the execution of an encrypted function. If not, the hypervisor injects the exception to the guest, thus delegating the exception handling to the operating system. If the hypervisor detects an attempt to execute an encrypted function, it locates the encrypted version of this function, which was loaded during the hypervisor's initialisation, in the *database*. Then, the hypervisor decrypts the function to one or more safe pages (see Section 2.5.3). Finally, the hypervisor makes some preparations and jumps to the decrypted function.

To understand the nature of the steps outlined in the previosu paragraph, we now turn to the virtual address space of the hypervisor. Figure 19 illustrates the virtual address space layouts of the hypervisor and the guest. In the x86-64 instruction set, code and memory accesses are instruction-relative, meaning that the same sequence of instructions produces different results if executed from different virtual addresses. It is therefore highly important to execute the decrypted functions from their natural virtual addresses. Usually operating systems divide the virtual address space into two large regions. In 64-bit Windows, the upper half of the 64-bit space contains the code and data of the kernel, while the lower half contains the code and data of a process. The hypervisor mimics this behaviour by holding its code and data in the upper half of its virtual address space, while the lower half is reserved for decrypted functions and their data. Whenever the hypervisor decrypts a function to a safe page, it maps this safe page such that the virtual address of the decrypted function equals the virtual address of the protected function. Finally, the hypervisor transitions to user-mode (under the context of the hypervisor) and jumps to the decrypted function. (These two operations are performed by a single `IRET` instruction.) The execution of the decrypted function continues until it generates an exception. The hypervisor can handle some exceptions, while others are injected to the operating system. During its execution, a decrypted function can attempt to read from or write to a page that is not mapped in the hypervisor's virtual address space. In the latter instance, the hypervisor copies the corresponding mapping from the operating system's virtual address space. When the decrypted function completes and the hypervisor returns to the guest, the mappings that were constructed are retained for future invocations of that function. However, the operating system is free to reorganise its mappings (e.g., due to paging), thus rendering the hypervisor's mapping invalid. The handling of different cases is described in Algorithm 6. The algorithm outlines the implementation of the hypervisor's VM-exit handler. Each VM-exit is caused by some condition that occurred in the guest (or during the guest's execution). The primary condition that the hypervisor intercepts is a general protection exception. General protection exceptions can occur either due to the execution of a `HLT` instruction or for some other reason. The hypervisor should provide a special handling only for the first case (lines 3–16); in the second case, the hypervisor should inject the exception into the operating system (lines 1–2).

Algorithm 6 Hypervisor's VM-exit handler

```
 1: if #GP and RIP is not in protected function then
 2:     Inject and return to guest
 3: else if #GP and RIP is in protected function then
 4:     while TRUE do
 5:         Enter user-mode at RIP and await interrupts
 6:         if #GP or #PF[INSTR] then
 7:             if RIP not in protected function then
 8:                 Return to guest
 9:             if RIP is not mapped then
10:                 Allocate a safe page & fill it with HLTs
11:                 Map the page to RIP
12:             Decrypt function at RIP
13:         else if #PF[DATA] and mapped in guest then
14:             Copy mapping
15:         else
16:             Inject and return to guest
17: else if INVLPG then
18:     Clear virtual table
19:     Return to guest
```

The hypervisor reacts to a special instruction-induced general protection exception by entering a loop (lines 4–16). At the beginning of each iteration (line 5), the hypervisor transitions to user-mode (without returning to the guest) and sets the instruction pointer to the same address that generated the general protection exception. The execution continues until an exception occurs in user-mode.

During execution of regular functions or handling of interrupts and exceptions, the operating system may modify the mappings of virtual pages. The hypervisor may have copies of some of these mappings, which were modified by the operating system. Thus, it is essential for the hypervisor to intercept all such modifications. Fortunately, according to Intel's specification [INTEL16], since the processor stores portions of mapping information in its caches (TLBs), the operating system is required to inform the processor of all modifications through a special instruction, `INVLPG`. The hypervisor intercepts this instruction and responds to it by erasing all the entries that were copied from the operating system (lines 17–19).

3.4.4 Analysis and Performance

The performance of the described system depends greatly on the set of encrypted functions. The amount of intellectual property contained within a program may vary, and as does the set of encrypted functions. We note that the performance is affected not only by the number of encrypted functions but also by the interconnections between these functions. This fact complicates the performance evaluation, since the functions to be encrypted are not known. Our evaluation is therefore in part randomised. The evaluation presented below answers the following questions:

1. How does our system compare to obfuscation with respect to performance?
2. What is the expected performance degradation when X% of a program is encrypted?
3. To what extent can initially poor performance be improved?

To answer the first question, we protected the same program using our system and using Obfuscator-LLVM and measured the performance overhead. To answer the second question, we performed a randomised experiment, during which function sets of different sizes were randomly selected for encryption. Finally, for the third question, we show that the hypervisor's built-in profiler can be used to improve initially poor performance by two orders of magnitude.

In the first test, we analyse the performance impact of Obfuscator-LLVM [JRW+15] compared to our method. We cloned the latest Obfuscator-LLVM directly from the official Git repository and built a 32-bit version. For the comparison, we protected 7-Zip using Obfuscator-LLVM and using our system. Specifically, we tested the 'extracting files from an archive (the `e` command line option)' and used a 7z compressed tarball of the latest Linux kernel to date (4.15.6).

We performed two tests that differed in the set of functions that was selected for protection. In the first test, the set included the functions `DecodeToDic, DecodeReal2, and WriteRem`, which constitute $\approx$ 1% of the total execution time. In the second test, we added `DecodeReal` to the set of functions, which then constituted $\approx$ 84% of the total execution time. In both tests, the functions were encrypted using our system and obfuscated using Obfuscator-LLVM with the following obfuscating transformations: instruction substitution (SUB), bogus control flow (BCF) and control flow flattening (FLA).

In the first test, Obfuscator-LLVM and our system both showed an overhead of $\approx$ 5%. In the second test, the overhead of our system was still $\approx$ 5%, while the overhead of Obfuscator-LLVM was $\approx 13,500\%$. Figure 20 presents the execution times of:

1. the original program.
2. the same program protected using our system.
3. the same program protected using Obfuscator-LLVM with SUB alone.
4. the same program protected using Obfuscator-LLVM with BCF alone.
5. the same program protected using Obfuscator-LLVM with FLA alone.
6. the same program protected using Obfuscator-LLVM with SUB, BCF, and FLA.

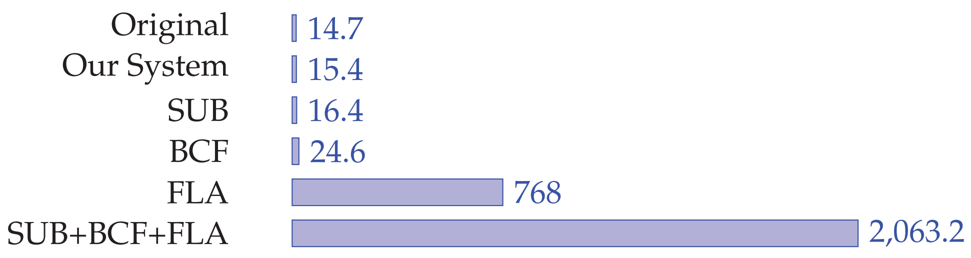


FIGURE 20 Execution times in seconds of the original, encrypted, and obfuscated versions, 7-Zip, in the second experiment.

The results are quite expected. Our system is not affected by the contents of the functions, or the number of times the function is called as long as the protected code mostly executes in the hypervisor.

In the second test, we chose a predetermined set S of functions that covers a significant part of the executable. For the encryption, we constructed 11 subsets of S: $S_0, S_1, \dots S_{10} \subseteq S$, where S_i consists of $\frac{i}{10}$ fraction of functions from S selected at random. The encryption resulted in 11 protected executables $P_0, P_1, \dots, P_{10}$, where P_0 is the original program and P_{10} consists of all the functions in S encrypted. Two executables were tested in the second experiment:

1. A 32-bit version of LAME MP3 Encoder [LAME] — we chose all the functions that belong to the *lame* namespace (which covers 56% of LAME's main executable functions). We ran LAME with the following three configurations:

   - fixed bit rate 128kbps encoding — default LAME behaviour.
   - fixed bit rate jstereo 128kbps encoding, high quality.
   - fast encode, low quality (no psycho-acoustics).

   results are depicted in Figure 21.

2. A 32-bit version of 7-Zip — our set S included all the functions of the LZMA algorithm. These functions lay within the '_Lzma' namespace. We analysed the decompression time of the latest Linux kernel to date (4.15.6). The results are depicted in Figure 22.

In the third test, we tested the impact of the built-in profiler on the performance of an encrypted program. The built-in profiler allows one to obtain a better view of the relationships between the encrypted and non-encrypted functions. This information might be critical, as every branching to a non-encrypted function requires two costly operations: a VM-entry and a VM-exit. As the number of transitions between encrypted and non-encrypted functions increases, so does the total execution time of the program. The built-in profiler records all the transitions from an encrypted to a non-encrypted function (step 8 in Algorithm 6). For each transition, the hypervisor stores the faulting address (i.e., the address of

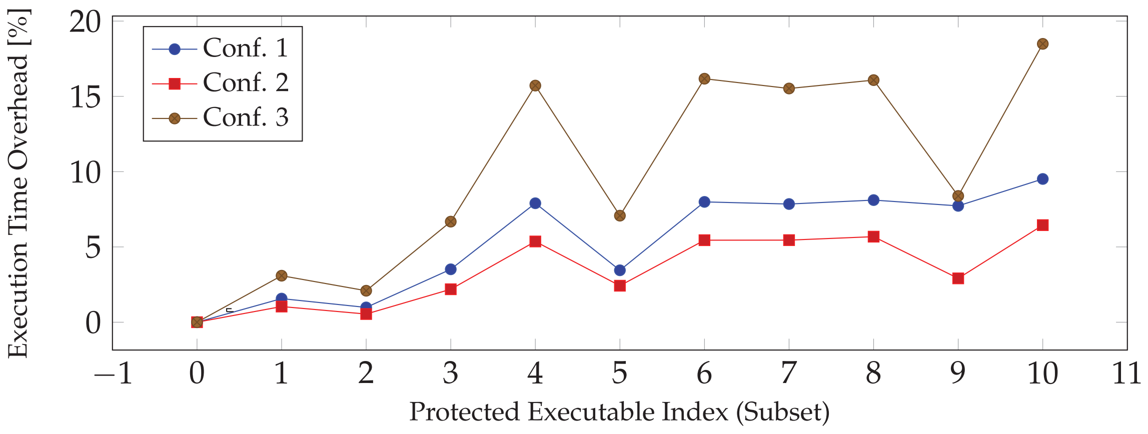


FIGURE 21 The overhead of protected executables. Each line represents a single configuration. A mark on a line represents an average execution time overhead (in percentages) of a protected executable compared to the original executable. Consider the mark inside the dashed square, which corresponds to P_4 when run in the first configuration. According to this mark, P_4 is slower than the original program (P_0) by 7.9%.

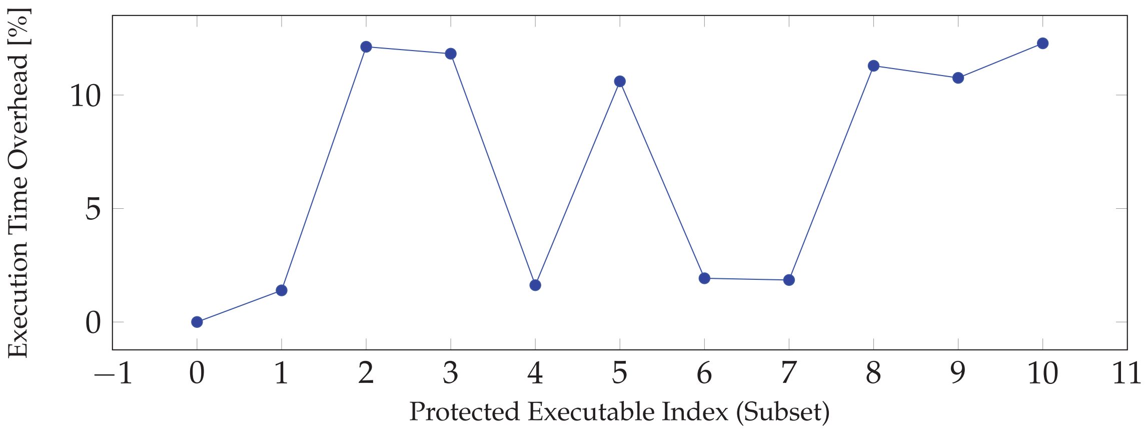


FIGURE 22 The overhead of protected executables. A mark on a line represents an average execution time overhead (in percentages) of a protected executable compared to the original executable.

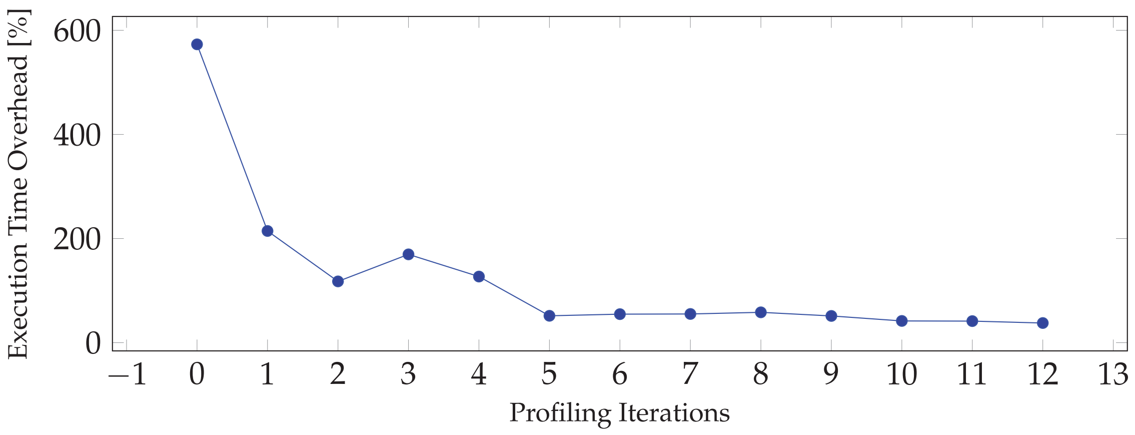


FIGURE 23 The overhead of the protected executable during profiling iterations. A mark at position *i* represents the overhead in percents of the protected executable compared to the original executable at the *i*th iteration. Note the exponential behaviour of the overhead improvement.

TABLE 1 Execution time (in milliseconds) of 7-Zip tests.

	Compression	Decompression
Original	122.7143	6.4979
Protected	122.7701	6.5617

the non-encrypted function) and the total number of transitions to this address that occurred during the current execution.

The dynamic analysis behaviour of the built-in profiler provides a great advantage over static analysis, as the control flow of a program frequently depends on its input and cannot be known in advance. However, a combination of the two analysis techniques can be used to increase the ease of profiling process. We tested a total of three applications:

1. *openssl* – specifically, we encrypted a set of functions in *libcrypto* and tested the RSA key generation functionality (*genrsa* command). Figure 23 depicts the results.

2. *7zip* – specifically, we tested two functionalities: (1) adding files to an archive (the ’a’ command) and (2) extracting files from an archive (the ’e’ command). Table 1 depicts the results.

3. *Apache Web Server* – As a starting point, we selected three functions, within `libhttpd` that are heavily used by Apache to handle HTTP requests. As our benchmark utility, we used the Apache Benchmark (`ab`) tool. We measured the time it took for the server to handle 20,000 requests (`-n` option in `ab`). Table 2 depicts the results.

TABLE 2 Execution time (in seconds) of Apache tests.

Case	Handling Time
Original	11.46
Iteration 0	17.7
Iteration 1	15.6
Iteration 2	13.8

4 SUMMARY OF ORIGINAL ARTICLES

In total, six articles are included in the thesis, and this chapter provides a summary of each, first addressing the research problem then describing the results that were obtained.

4.1 Hypervisor-Based White Listing of Executables

R. Leon, M. Kiperberg, A.A. Leon Zabag, A. Resh, A. Algawi, N. Zaidenberg;, "Hypervisor-Based White Listing of Executables", IEEE Security & Privacy, 2019.

4.1.1 Research Problem

This article studies current methods for preventing unauthorised execution. These methods can be divided into three categories [IM07]: (1) behavioural, (2) signature-oriented, and (3) whitelist-based. Behavioural systems are based on behaviour analysis and are therefore difficult to evaluate while signature-oriented systems are prone to zero-day attacks. Therefore, particularly in systems that do not change frequently, whitelist-based systems are often the preferred solution.

Typically, whitelist enforcement is performed by intercepting executable images at their load time (e.g., by intercepting system calls) [F17]. However, in the event that there is a vulnerability, exploitation becomes possible in runtime [B18]. Furthermore, current whitelist-based systems use a service application or a kernel application to perform the enforcement, thus leaving them exposed to vulnerabilities in the operating system itself.

This article presents a system that allows native executable images (both kernel and user applications) to be monitored during their entire lifetime. The system provides strong security guarantees and induces a negligible performance overhead.

4.1.2 Results

This article presents a detailed description of a whitelist-based system and claims that the suggested method does not suffer from the two main deficiencies present in current methods for the following reasons:

1. The execution of a given executable image (both in user mode and kernel mode) is enforced during its entire lifetime.

2. The system can prevent unauthorised code from being executed even in the event that an attacker has full control over the operating system kernel.

The article describes the preparations, initialisation, and operation of the system and presents an executable scanner and a thin hypervisor. The executable scanner creates a whitelist-database by scanning an initially trusted system, scanning both kernel and user applications in the same manner, while a thin hypervisor, using the whitelist-database, is used to enforce authorised execution.

Next, the article discusses the security of the proposed system and how it fulfils the given security guarantees before finally presenting an evaluation of the proposed system. The results suggest that the performance overhead of the system is negligible.

4.2 Hypervisor-assisted Atomic Memory Acquisition in Modern Systems

M. Kiperberg, R. Leon, A. Resh, A. Algawi, N. Zaidenberg;, "Hypervisor-assisted Atomic Memory Acquisition in Modern Systems", ICISSP Conference, 2019.

4.2.1 Research Problem

This article studies the problem of acquiring an atomic memory image. In many cases, the analysis of an attack is divided into two steps: memory acquisition and static analysis. In the first step, a software [QXM17] or a hardware tool [ZWZ+10] acquires the memory contents of the running system and stores it for later analysis. In the second step, a static analysis tool (e.g., [C14, PWF+06]) is applied to the acquired image of memory to analyse the malicious software.

The article discusses software (hypervisor-based) solutions that can acquire an atomic memory image. These methods work similarly; however, the described methods are limited to a single execution unit and were tested on Windows XP SP3.

This article presents a system that provides atomic memory acquisition capabilities. However, we show how the system can be implemented on modern systems that have multiple execution units and modern security features.

4.2.2 Results

This article presents two problems that arise with current hypervisor-based methods. The first problem is the availability of multiple processors. Each processor has direct access to the main memory and can freely modify any page, so when the hypervisor is requested to start memory acquisition, it must therefore configure all memory pages on all processors to be non-writable. The second problem is the delay sensitivity of some memory pages. We have observed that, in modern systems, some memory pages cannot be blocked for writing, as the access time to these pages is critical. This article proposes two solutions to the these problems.

The article presents a detailed description of the memory acquisition process, suggesting the use of a documented functionality (which requires no operating system kernel patches) to initiate the memory acquisition process on the secondary processors. Furthermore, it discusses how the location of the sensitive memory pages can be obtained on Windows 10, regardless of ASLR. Finally, the article presents an evaluation that includes the memory usage, and memory acquisition performance of the system. The memory usage of the system depends on the number of delay-sensitive memory pages. We could achieve a maximum speed of 97920KB/s, at which speed the system became unresponsive.

4.3 Hypervisor-based Protection of Code

M.Kiperberg, R. Leon, A. Resh, A. Algawi, N. Zaidenberg;, "Hypervisor-based Protection of Code", IEEE Transactions on Information Forensics and Security, 2019.

4.3.1 Research Problem

This article studies the problem of software reverse-engineering. Current countermeasures to reverse-engineering are based on obfuscation. Generally, obfuscation methods suffer from two primary deficiencies: (1) the obfuscated code is less efficient than the original; and (2) with sufficient effort, the original code can be reconstructed. Using obfuscation, the program is transformed into a more complex program which nevertheless remains semantically identical. A wide range of obfuscation transformations exist, such as instruction reordering, and instruction substitution. However, these transformations are vulnerable to automatic attacks [ROL09].

Another completely different approach is encoding, which either encrypts or compresses portions of the original program and decodes these portions back before their execution. However, even when encryption is used, the decryption key is stored within the decryption algorithm and can therefore be extracted.

The performance degradation of obfuscation methods depends on their sophistication. For example, the LLVM-Obfuscator [JRW+15] specifies which obfuscation techniques should be applied to an executable. We have shown that in the event that several obfuscation techniques are in use, the execution times increase by a factor of 15-35.

4.3.2 Results

This article presents a detailed description of a system that allows native programs to be encrypted and executed. The article claims that the suggested method can withstand the following types of attacks:

- malicious code executing in user-mode or kernel-mode.
- malicious hardware devices connected via a DMA controller equipped with IOMMU.
- sniffing on any bus.

The article describes the preparations, initialisation, and operation of the system and presents an encryption tool, a thin hypervisor, and a built-in profiler. The encryption tool encrypts functions of a program. The program can then be executed as a regular program, provided that the proposed system is installed. The thin hypervisor is initialised such that the code, once decrypted, cannot leave the confines of the CPU; that is, it is guaranteed never to leave the cache. Upon an execution attempt of an encrypted function, a transition to the thin hypervisor occurs. The thin hypervisor decrypts the function and executes it, but stops the decrypted code execution in two cases: (1) a non-encrypted function was called or (2) an interrupt occurred. The built-in profiler allows the performance of an encrypted program to be improved.

The article discusses the security of the proposed system and how it fulfils the given security guarantees. Finally, the article presents an evaluation of the proposed system which consists of (1) a comparison with LLVM-obfuscator, (2) a randomised experiment, and (3) usage of the built-in profiler. The evaluation of the system's efficiency suggests that it can compete with and outperform obfuscation-based methods.

4.4 Preventing Execution of Unauthorized Native-Code Software

A. Resh, M. Kiperberg, R. Leon, N. Zaidenberg;, "Preventing Execution of Unauthorized Native-Code Software," International Journal of Digital Content Technology and its Applications (JDCTA), 2017.

4.4.1 Research Problem

This article studies the problem described in Paper [PI]. However, because the described system uses a type 2 hypervisor (i.e., hosted hypervisor) this paper also studies the problem of remote computer authentication [KRZ15], a process by which one computer determines whether another computer is running the correct version of the software. A service provider that possesses a remote authentication method can enforce authorised-only access to a remote resource, for example, gaming consoles that are allowed to connect to gaming networks only if authenticated [I09].

4.4.2 Results

This article presents a detailed description of a system that enforces authorised code execution, describing three components: (1) execution database, (2) management station, and (3) thin hypervisor. The execution database contains information regarding every executable image in the filesystem, including, among others, the hashes of every single executable page. The management station has two responsibilities: (1) authenticating the machine before initialising the hypervisor and (2) monitoring the system (e.g., providing alerts regarding execution violations). Once authenticated, a thin hypervisor is deployed on the machine. The thin hypervisor is responsible for enforcing authorised execution using the execution database.

4.5 System for Executing Encrypted Native Programs

A. Resh, M. Kiperberg, R. Leon, N. Zaidenberg;, "System for Executing Encrypted Native Programs," International Journal of Digital Content Technology and its Applications (JDCTA), 2017.

4.5.1 Research Problem

This article studies the problem described in Paper [PIII]. However, because the described system uses a type 2 hypervisor (i.e., a hosted hypervisor) this paper also studies the problem of remote computer authentication (see Section 4.4.1). The method presented in Paper [PIII] provides better security guarantees along with better performance.

4.5.2 Results

This article presents a detailed description of a system for executing native programs. The article describes three components: (1) the encryption tool, (2) the attestation server, and the (3) thin hypervisor. The encryption tool encrypts the functions of a program, which can then be executed as a regular program, provided that the proposed system is installed. The attestation server is responsible for authenticating the hypervisor, and once the latter is authenticated, the attestation server delivers the hypervisor a decryption key, which is stored within internal unused processor registers. Upon an execution attempt of an encrypted function, a transition to the thin hypervisor occurs, and the thin hypervisor decrypts the function and executes it. The article presents two execution methods: in-place execution and buffered-execution. The former is more secure while the latter is more efficient. Paper [PIII] implements the buffered-execution method while overcoming the security issues presented in this paper.

4.6 Hypervisor-assisted Dynamic Malware Analysis

R.Leon, M. Kiperberg, A.A. Leon Zabag, N. Zaidenberg;, "Hypervisor-assisted Dynamic Malware Analysis", ACM Transactions on Privacy and Security (TOPS), Submitted

4.6.1 Research Problem

This article studies the problem of dynamic malware analysis. There are two approaches for malware analysis: a static and a dynamic approach. Malware use an abundance of evasion techniques to avoid detection by static tools. Therefore, dynamic analysis is often the preferred approach. The dynamic analysis approach often involves executing the malware and recording its system calls. Current dynamic malware analysis methods can be classified into four categories: (1) hooking methods, (2) emulation methods, (3) hypervisor-based methods, and (4) bare-metal based methods. Hooking methods [WHF07, GF10, YMH+17] are highly efficient but are detectable. Emulation methods are extremely slow but are more difficult to detect (yet, are prone to incomplete emulation [F06, F07, RKK07, DRS+08, VC14]). Current hypervisor-based methods are more efficient than emulation-based methods, but the performance overhead they cause is significant ([DRS+08, LMP+14]). Bare-metal methods are highly efficient and difficult to detect, but their ability to generate an informative behavioural profile is lacking.

4.6.2 Results

Our method can be classified as a hypervisor-based method. However, in our method:

1. The system call monitoring component is located within the guest operating system and consequently no intervention of the hypervisor is required during the system call monitoring and analysis. As a result, the performance overhead is kept to a minimum.
2. Though completely handled in the guest context, the monitoring component is fully transparent to the guest operating system.

We demonstrate how the hypervisor is used to hide the presence of the monitoring component. The performance overhead of our system is negligible, as the hypervisor is not involved in the system call monitoring process. The information extracted by our method can be used to build an informative behavioural profile for the monitored process.

5 CONCLUSIONS

In this chapter, we summarise the contributions of the thesis, discuss the limitations of the research reported herein, and outline some directions for further research.

5.1 Contributions

The thesis presents applications of hypervisors in security, specifically presenting four common security problems, the solutions to which share a common architecture. Each security problem extends the common architecture with its necessary components. The common architecture of the system combines several components, some of which were previously studied and others of which are novel. The thesis describes the novel components and extends the studied components.

This thesis extends the current hypervisor-based solution to memory image acquisition, which was studied by [MFP+10, QXM17]. The extension adapts this method to modern operating systems and modern processors, which have multiple execution units.

This thesis extends current hypervisor-based solutions to dynamic malware analysis which have been studied by [DRS+08, LMP+14], and presents an efficient, evasive, and effective way to perform dynamic malware analysis. The thesis presents a novel approach, in which a monitoring component is injected into the address space of the operating system and the hypervisor protects the monitoring component from modification and detection. This approach allows system performance to be maximised.

This thesis extends current solutions to unauthorised execution prevention studied by [SLQ07, RKL+17]. The proposed system is whitelist-based [M09] and provides a solution which grants real-time page granularity execution protection for both kernel and user applications. The performance overhead of the system is negligible.

Finally, the thesis extends current solutions to reverse engineering prevention. Current solutions, as described, are mainly based on obfuscation, while the described system is based on encoding. However, the provided security guarantees are comparable to those given by obfuscation-based methods. The thesis extends hypervisor-based encryption solution which was studied by [AKZ11, KRA+17]. The decryption key and the decrypted code never leave the confines of the processor, and are therefore inaccessible to even an adversary with bus-sniffing capabilities. The described system can compete with and outperform obfuscation-based methods.

5.2 Limitations and Future Research

This section presents the limitations of the described systems and outlines directions for further research.

5.2.1 Thin Hypervisor Transparency

As described in Section 2.6, a hypervisor may need to protect itself (and, possibly, its actions) from detection. The guest operating system is unaware of the hypervisor's intervention, as the former is given the illusion that it executes directly on the hardware. However, there is a time-toll for the hypervisor mediation [BB05, GAW+07]. This section presents an approach with which the hypervisor can protect its actions, specifically, memory modifications, from detection.

The question of whether the hypervisor itself can be detected remains open [FLM+07, GAW+07, RT07, RT08, BYDD+10, BBN12]. Common detection attacks involve local timers, and these attacks can be dealt with quite easily, as they typically use the `RDTSC` instruction, which can be intercepted by the hypervisor (other local timers can be dealt with similarly). We plan to research this subject in the near future.

5.2.2 Unauthorised Execution Prevention

As described in Section 3.3, the described system provides a whitelist-based solution for native executable images.

In recent years, programs that are targeted at managed execution environments have become widespread. Managed programs, unlike native ones, cannot be executed directly by the CPU and are typically executed by a native program, which benefits from our system, and any unknown execution attempt on its behalf will be blocked. However, the programs it executes do not benefit from our system.

This section presents an approach for protecting managed programs. We plan to research this approach in the near future to improve the scalability of the system.

Attacks that are based on existing code have become more popular, as many countermeasures exist for attacks that are based on code injection. Our method assures that only signed code is allowed to execute. Therefore, attacks that use known-existing code are not blocked.

This section suggests integrating existing countermeasures for code-reuse attacks into our system. We plan to perform this integration and research its overall impact and effectiveness in the near future.

5.2.3 ARM Architecture

The security solutions presented in this thesis can be ported to the ARM architecture on ARM devices (e.g., most of today's smartphones) that implement the virtualization extensions. ARM virtualization extensions provide capabilities similar to Intel VT-x. For example, they provide a mechanism similar to Intel EPT for guest-physical-address (IPA in ARM terminology) to host-physical-address translation. The ARM Security Extensions, known as TrustZone, provide a way to create an isolated environment in which sensitive applications can execute. This isolated environment executes at the highest privilege level (higher than the hypervisor) and is not subject to virtualization. Due to the latter, for better security guarantees, our system might use TrustZone in addition to a hypervisor.

YHTEENVETO (FINNISH SUMMARY)

Tämä väitöskirja esittelee hypervisorien sovelluksia tietoturvallisuudessa, ja erityisesti se esittelee neljä yhteistä turvallisuusongelmaa, joiden ratkaisuun käytetään yhteistä arkkitehtuuria. Jokainen tietoturvaongelman ratkaisu laajentaa yhteistä arkkitehtuuria tuomalla mukaan siihen tarvittavat komponentit. Järjestelmän yhteinen arkkitehtuuri yhdistää useita eri komponentteja, joista osa on aiemmin tutkittu ja toiset ovat aivan uusia.

Tämä työ laajentaa nykyisen hypervisori-pohjaisen ratkaisun muistissa olevan datan hankkimiseen. Laajennus mahdollistaa tämän menetelmän käytön nykyaikaisiin käyttöjärjestelmiin ja prosessoreihin, joilla on useita suoritusyksiköitä. Väitöskirja mahdollistaa nykyisten hypervisor-pohjaisten ratkaisujen käytön dynaamisten haittaohjelmien analyysin ja antaa tehokkaan tavan suorittaa dynaaminen haittaohjelman analyysi. Väitöskirja esittelee uudenlaisen lähestymistavan, jossa valvontakomponentti sijoitetaan käyttöjärjestelmään ja hypervisorin suojaa monitoroivaa komponenttia muokkauksilta ja havaitsemiselta. Tämä lähestymistapa mahdollistaa järjestelmän suorituskyky maksimoinnin. Väitöskirja laajentaa myös nykyiset ratkaisut luvattoman suorituksen estämiseen. Esitetty menetelmä on sallittuihin luetteloihin perustuva ja tarjoaa ratkaisun, joka antaa reaaliaikaisen suoritussuojan sekä ytimen että käyttäjän sovelluksille. Menetelmän vaikutus koko järjestelmän suorituskyvylle on hyvin pieni.

Lopuksi tutkielma laajentaa nykyisiä ratkaisuja käänteisen suunnittelun estämiseen. Nykyiset ratkaisut perustuvat pääasiassa hämärtämiseen, kun taas kuvattu järjestelmä perustuu koodaukseen. Esitettyjen ratkaisuiden turvallisuustakuut ovat verrattavissa hämärtämispohjaisiin menetelmiin. Väitöskirjan tulokset laajentavat hypervisor-pohjaista salausratkaisua. Salausavain ja salattu koodi eivät koskaan jätä rajapintoja prosessoriin päin ja ovat siten näkymättömiä väylää tutkiville haittaohjelmille. Esitetyt ratkaisut toimivat tehokkaammin kuin hämärtämiseen perustuvat menetelmät.

REFERENCES

[AD10] Aumaitre D., Devine C., "Subverting Windows 7x64 Kernel with DMA attacks", *HITBSecConf 2010 Amsterdam*, Vol. 29, 2010.

[AKL+19] Algawi A., Kiperberg M., Leon R., A Resh and Zaidenberg N., "Creating Modern Blue Pills and Red Pills", in *European Conference on Cyber Warfare and Security* pp. 6-14, 2019

[AKL+19b] Algawi A., Kiperberg M., Leon R., A Resh and Zaidenberg N., "Using Hypervisors to Overcome Structured Exception Handler Attacks", in *European Conference on Cyber Warfare and Security* pp. 1-5, 2019

[AKL+19c] Algawi A., Kiperberg M., Leon R., A Resh and Zaidenberg N., "Efficient Protection for VDI Workstations", in *IEEE CSCloud*, 2019.

[AKL+20] Algawi A., Kiperberg M., Leon R., A Resh and Zaidenberg N., "Modern Blue Pills and Red Pills", accepted in *Encyclopedia of Criminal Activities and the Deep Web*, 2020.

[AKZ11] Averbuch A., Kiperberg M., and Zaidenberg N., "An efficient vm-based software protection", in *Network and System Security (NSS), 2011 5th International Conference on, pp. 121-128.*, IEEE, 2011.

[AKZ13] Averbuch A., Kiperberg M., and Zaidenberg N.J., "Truly-protect: An efficient vm-based software protection", *IEEE Systems Journal*, 2013, 7 (3), 455-466

[AMD18] "AMD64 Architecture Programmer's Manual Volume 2: System Programming", 2018.

[ARM03] "ARM Limited. ARM builds security foundation for future wireless and consumer devices", ARM Press Release, May 2003.

[ARM10] "Virtualization Extensions Architecture Specification", 2010.

[B18] Beechy J., "Application Whitelisting: Panacea or Propaganda", https://www.sans.org/reading-room/whitepapers/application/application-whitelisting-panacea-propaganda-33599, 2010, [Online; accessed September-2019].

[BB05] Brumley D., Boneh D., "Remote timing attacks are practical", *Computer Networks*, 48(5), pp. 701-716, 2005.

[BBN12] Branco R.R., Barbosa G.N., and Neto P.D., "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm techniques", *Blackhat USA '12*, 2012.

[BCI+15] Bianchi A., CorbettaJ., Invernizzi L., Fratantonio Y., Kruegel C., Viana G., "What the App is That? Deception and Countermeasures in the Android User Interface", in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2015.

[BDF+03] Barham P., Dragovic B., Fraser K., Hand S., Harris T., Ho A., Neugebauer R., Pratt I., and Warfield A., "Xen and the art of virtualization", in *Proc. 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, USA, SOSP 2003.

[BKK06] Bayer U., Kruegel C., and Kirda E., "TTanalyze: A Tool for Analyzing Malware", in *EICAR*, pp. 180-192, 2006.

[BLS13] Basya D., Low R., and Stamp M., "Structural entropy and metamorphic malware", *Journal of Computer Virology and Hacking Techniques*, 9(4):179-192, 2013.

[BS96] Bressoud T. and Schneider F.B., "Hypervisor-based fault tolerance.", *ACM Trans. Comput. Syst.*, 14(1), pp. 80-107, 1996.

[BYDD+10] Ben-Yehuda M., D Day M., Dubitzky Z., Factor M., Har'El N., Gordon A., Liguori A., Wasserman O., and Yassour B., "The Turtles Project: Design and Implementation of Nested Virtualization", in *OSDI*, volume 10, pp. 423-436, 2010.

[BYLZ19] Ben Yehuda R., Leon R. and Zaidenberg N., "ARM Security Alternatives", in *European Conference on Cyber Warfare and Security*, pp. 604-612, 2019.

[BYZ18] Ben Yehuda R. and Zaidenberg N., "Hyplets-Multi Exception Level Kernel towards Linux RTOS", in Proceedings of the *11th ACM International Systems and Storage Conference* 2018.

[BYZ19] Ben Yehuda R. and Zaidenberg N., "Protection Against Reverse Engineering in ARM", in *International Journal of Information Security*, 2019.

[BZN11] Burguera I., Zurutuza U., and Nadim-Therani S., "Crowdroid: behavior-based malware detection system for Android", in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, NY, USA, SPSM '11, ACM., pp. 15-26, 2011.

[C14] Cohen M., "Rekall memory forensics framework", in *DFIR Prague 2014*, SANS DFIR, 2014.

[CAM+08] Chen X., Andersen J., Mao Z.M., Bailey M., and NazarioJ., "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware", in *Proc. IEEE Conf. DSN*, pp. 177-186, June 2008.

[CG04] Carrier B. D. and Grand J., "A hardware-based memory acquisition procedure for digital investigations", *Digital Investigation* 1(1)50-60, 2004.

[DB17] Durve R. and Bouridane A., "Windows 10 security hardening using device guard whitelisting and applocker blacklisting", in *Emerging Security Technologies (EST), 2017 Seventh International Conference*, on pp. 56-61, 2017.

[DPM02] Denys G., Piessens F., and Matthijs F., "A survey of customizability in operating system research", *ACM Computing Surveys*, pp. 450-468, 34(4), 2002.

[DRS+08] Dinaburg A., Royal P., Sharif M.I., and Lee W., "Ether: malware analysis via hardware virtualization extensions", in *proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 51-62.

[DWH+12] Ding B., Wu Y., He Y., Tian S., Guan B., and Wu G., "Return-oriented programming attack on the xen hypervisor", in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference*, pp. 479-484, IEEE, 2012.

[F06] Ferrie P., "Attacks on virtual machine emulators", *Symantec Advance Threat Research*, 2006.

[F07] Ferrie P., "Attacks on more virtual machine emulators", *Symantec Technology Exchange*, 2007.

[F17] Fanton et al., "Secure System For Allowing The Execution of Authorized Computer Program Code", *U.S. Patent No. 9,665,708 B2, May 30, 2017*.

[FLM+07] Franklin J., Luk M., McCune J.M., Seshadri A., Perrig A., van Doorn L., "Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking", *Carnegie Mellon CyLab (2007)*, 2007.

[GAW+07] Garfinkel T., Adams K., Warfield A., and Franklin J., "Compatibility Is Not Transparency: VMM detection myths and realities", in *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[GBS14] Gandotra E., Bansal D., and Sofat S., "Malware Analysis and Classification: A Survey", *Journal of Information Security*, pp. 56-64, 5(02), 2014.

[GF10] Guarnieri C. and Fernandes D., "Cuckoo Automated Malware Analysis System", https://cuckoosandbox.org/, February 2010, [Online; accessed September-2019].

[I09] Ionescu D., "Microsoft bans up to one million users from xbox live", *PC World*, 2009.

[IM07] Idika N. and Mathur A.P., "A survey of malware detection techniques", Purdue University 48, 2007.

[INTEL16] "Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3", 2016.

[IntelIM18] "Intel Virtualization Technology for Directed I/O Architecture Specification", 2018.

[JO05] Jonsson E., Olovsson T., "A quantitative model of the security intrusion process based on attacker behavior", *IEEE Transactions on Software Engineering*, 24(3):235-45, April 1997.

[JRW+15] Junod P., Rinaldini J., Wehrli J., and Michielin J., "Obfuscator-LLVM – software protection for the masses", in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*, SPRO'15, Firenze, Italy, May 19th, 2015.

[KLR+19] Kiperberg M., Leon R., Resh A., Algawi A. and Zaidenberg N.J., "Hypervisor-Based Protection of Code", in *IEEE Transactions on Information Forensics and Security*, 14 (8):2203-2216, 2019.

[KLR+19b] Kiperberg M., Leon R., Resh A., Algawi A. and Zaidenberg N., "Hypervisor-assisted Atomic Memory Acquisition in Modern Systems", in *5th International conference on Information Systems Security and Privacy*, 2019.

[KRA+17] Kiperberg M., Resh A., Algawi A., and Zaidenberg N., "System for Executing Encrypted Java Programs", in *ICISSP Conference*, 2017.

[KRZ15] Kiperberg M., Resh A., and Zaidenberg N., "Remote Attestation of Software and Execution Environment in Modern Machine", in *CSCloud*, 2015.

[KVK11] Kirat D., Vigna G., and Kruegel C., "BareBox: efficient malware analysis on bare-metal", in *ACSAC. ACM*, December 2011.

[KVK14] Kirat D., Vigna G., and Kruegel C., "BareCloud: are-metal Analysis-based Evasive Malware Detection", in *23rd USENIX Security Symposium (USENIX Security 14). USENIX Association*, 2014.

[KZ13] Kiperberg M. and Zaidenberg N. "Efficient Remote Authentication", in *Journal of Information warfare*, 2013 12 (3), 49-55

[KZA11] Khen E., Zaidenberg N.J. and Averbuch A. "Using virtualization for online kernel profiling, code coverage and instrumentation", in *International Symposium on Performance Evaluation of Computer & Telecommunication Systems*, 2011.

[KZA+13] Khen E., Zaidenberg N.J., Averbuch A. and Fraimovitch E. "LgDb 2.0: Using Lguest for kernel profiling, code coverage and simulation", in *International Symposium on Performance Evaluation of Computer & Telecommunication Systems*, 2013.

[L11] Langner R., "Stuxnet: Dissecting a cyberwarfare weapon", *IEEE Security Privacy*, vol. 9 no. 3, pp. 49-51, May-Jun. 2011.

[LAME] http://lame.sourceforge.net/, 2018, [Online; accessed Feb-2018].

[LKLZ+19] Leon R., Kiperberg M., Leon Zabag A. A., Resh A., Algawi A. and Zaidenberg N. J., "Hypervisor-Based White Listing of Executables", in *IEEE Security & Privacy*, Vol. 11, No. 5, pp. 58-67, 2019.

[LMP+14] Lengyel T.K., Maresca S., Payne B. D., Webster G. D., Vogl S., and Kiayias A., "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system", in *proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)(2014)*, 2014.

[LMS+11] Long F., Mohlndra D., Seacord R.C., Sutherland D.F., and Svoboda D., "The CERT Oracle Secure Coding Standard for Java", *Carnegie Mellon Software Engineering Institute (SEI) series, Addison-Wesley*, 2011.

[LPT07] Lambrinoudakis C., Pernul G., Tjoa M., *Trust, Privacy and Security in Digital Business: 4th International Conference, TrustBus 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, 2007.

[LT18] M. Larabel and M. Tippett, "Phoronix test suite", http://www.phoronix-test-suite.com/, [Online; accessed September-2019].

[LWJ+13] Li J., Wang Q., Jayasinghe D., Park J., Zhu T., and Pu C., "Performance Overhead Among Three Hypervisors: An Experimental Study using Hadoop Benchmarks", *IEEE International conference on Big Data*, pp. 9-16, 2013.

[M09] Steve Mansfield-Devine, "The promise of whitelisting", *Network Security*, Volume 2009, Issue 7, pp. 4-6, July 2009.

[M11] Morris T., "Trusted platform module", in *Encyclopedia of cryptography and security*. pp. 1332-1335, Springer, 2011.

[MFP+10] Martignoni L., Fattori A., Paleari R., and Cavallaro L., "Live and trustworthy forensic analysis of commodity production systems", in *International Workshop on Recent Advances in Intrusion Detection*, pp. 297-316, 2010.

[MMH08] Murray D.G., Milos G., and Hand S., "Improving Xen security through disaggregation", in *Proceedings of VEE'08*, pp. 151-160, NY, USA, 2008.

[NC05] Nanda S. and Chiueh T., "A survey on virtualization technologies", *Tech. Rep.*, 2005.

[OSM11] O'Kane P., Sezer S., and McLaughlin M., "Obfuscation: the hidden malware", IEEE Security & Privacy, Vol. 9, no. 5, pp. 41–47, 2011.

[PG74] Popek G. and Goldberg R., "Formal requirements for virtualizable third generation architectures", *Commun. ACM*, vol 17, pp. 412-421, July 1974.

[PJN10] Pilli E.S., Joshi R.C., and Niyogi R., "Network forensic frameworks: Survey and research challenges", *Digital Investigation*, pp. 14-27, 7(1), 2010.

[PRE12] Pareek H., Romana S., and Eswari P. R. L., "Application whitelisting: approaches and challenges", *International Journal of Computer Science, Engineering and Information Technology (IJCSEIT)*, Vol.2, No.5, 2012.

[PWF+06] Petroni Jr. N.L., Walters A., Fraser T., and Arbaugh W.A., "Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory", *Digital Investigation*, 3(4):197-210, 2006.

[QXM17] Qi Z., Xiang C., Ma R., Li J., Guan H., and Wei D. S. L., "ForenVisor: A tool for acquiring and preserving reliable data in cloud live forensics", *IEEE Trans. Cloud Comput*. 5, 3, 443-456, 2017.

[R06] Rutkowska J., "Subverting vista kernel for fun and profit", in *Proceedings of Black Hat*, USA, 2006.

[R17] R. Jason, "Columbo: High performance unpacking", in *IEEE 24th International Conference on Software Analysis , Evolution and Reengineering, SANER 2017*, Klagenfurt, Austria, February 20-24, 2017, pp. 507-510, 2017.

[RK07] Reiser H.P. and Kapitza R., "Hypervisor-based efficient proactive recovery", in *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, pp. 83-92, Oct. 2007.

[RKK07] Raffetseder T., Kruegel C., and Kirda E., "Detecting system emulators", *Information Security*, pp. 1-18, 2007.

[RKL+17] Resh A,, Kiperberg M, Leon R. and Zaidenberg N.J., "Preventing Execution of Unauthorized Native-Code Software", in *International Journal of Digital Content Technology and its Applications* 11, 2017.

[RKL+17b] Resh A., Kiperberg M., Leon R. and Zaidenberg N.J., "System for Executing Encrypted Native Programs", in *International Journal of Digital Content Technology and its Applications* 11, 2017.

[RMI12] Rad B., Masrom M., and Ibrahim S., "Camouflage in Malware: From Encryption to Metamorphism", *International Journal of Computer Science and Network Security*, Security, 12: 74-83, 2012.

[ROL09] Rolles R., "Unpacking Virtualization Obfuscators", in *Proceedings of the 3rd USENIX Conference n Offensive Technologies*, WOOT'09 pp. 1-1, USENIX Association, Berkeley, CA, USA, 2009.

[RT06B] Rutkowska J. "Introducing the Blue Pill", 2006, http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html, [Online; accessed September-2019]

[RT07] Rutkowska J. and Tereshkin A., "IsGameOver(), anyone?", in *Blackhat 2007*, 2007.

[RT08] Rutkowska J. and Tereshkin A., "Bluepilling the xen hypervisor", *Black Hat USA*, 2008.

[RZ13] Resh A. and Zaidenberg N.J., "Can keys be hidden inside the CPU on modern windows host", in *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*

[S08] Seacord R., "The CERT C secure coding standard", *Addison-Wesley Professional*, 2008

[SER12] Shabtai A., Elovici Y., and Rokach L., "A survey of data leakage detection and prevention solutions", *Springer Science & Business Media*, 2012.

[SET+09] Shinagawa T., Eiraku H., Tanimoto K., Omote K., Hasegawa S., Horie T., Hirano M., Kourai K., Ohyama Y., Kawai E., Kono K., Chiba S., Shinjo Y., and Kato K. "BitVisor: A thin hypervisor for enforcing I/O device security", in *Proceedings of the 5th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, ACM, 2009.

[SLQ07] Seshadri A., Luk M., Qu N., and Perrig A., "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity or commodity OSes", in *ACM SOSP*, 2007.

[SK10] Steinberg U. and Kauer B., "NOVA: A microhypervisor-based secure virtualization architecture", in *European Conference on Computer Systems*, April 2010.

[SSW+14] Sun H., Sun K., Wang Y., Jing J., and Jajodia S., "Trustdump: Reliable memory acquisition on smartphones", in *Proc. European Symposium on Research in Computer Security*, 2014.

[SVN+04] Shapiro J.S., Vanderburgh J., Northup E., Chizmadia D., "Design of the EROS Trusted Window system", in *Usenix Security*, 2004.

[TKM02] Tan. K. M. C., Killourhy K.S., and Maxion R. A., "Undermining an anomaly-based intrusion detection system using common exploits", *RAID*, 2002.

[UEFI] UEFI, "Unified Extensible Firmware Interface (UEFI) Specification", May 2019.

[UninformedKPP] Skywing "PatchGuard Reloaded A Brief Analysis of PatchGuard Version 3", http://uninformed.org/index.cgi?v=8&a=5, [Online; accessed September-2019].

[VMware] VMWare, "VMware ESX server virtual infrastsructure node evaluator's guide", https://www.vmware.com/pdf/esx_vin_eval.pdf, [Online; accessed September 2019].

[VC14] Vidas T. and Christin N., "Evading Android runtime analysis via sandbox detection", in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS '14)*, pp. 447-458, 2014.

[VF11] Vomel S., Freiling F., "A survey of main memory acquisition and analysis techniques for the windows operating system", *The International Journal of Digital Forensics & Incident Response*, 8(1), pp. 3-22, 2011

[W03] Whitman, M., "Enemy at the gate: threats to information security", *Communications of the ACM*, 46(8):91-95, 2003.

[WHF07] Willems C., Holz T., and Freiling F., "CWSandbox: Towards automated dynamic binary analysis", *IEEE Security & Privacy* 5(2), 2007.

[WJ10] Wang Z. and Jiang X., "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity", in *the 31th IEEE Symposium on Security and Privacy*, Oakland, 2010.

[XHT07] Ge X., Vijayakumar H., and Jaeger T., "Sprobes: Enforcing kernel code integrity on the trustzone architecture", in *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST 2014)*, May 2014.

[YHB+11] Younge A.J., Henschel R., Brown J.T., Von Laszewski G., Qiu J., and Fox G.C., "Analysis of virtualization technologies for high performance computing environments", *International Conference on Cloud Computing*, 2011.

[YMH+17] Yalew S.D., McGuire G., Haridi S., and Correia M., "T2Droid: A TrustZone-based dynamic analyser for Android applications", in *Proceedings of the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 25-36, Aug. 2017.

[YMM+17] Yalew S.D., Mendonça P., McGuire G., Haridi S., and Correia M., "TruApp: A TrustZone-based Authenticity Detection Service for Mobile Apps", in *Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing*, Networking and Communications (WiMob), 2017.

[YY10] You I. and Yim K., "Malware Obfuscation Techniques: A Brief Survey," Fifth International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA 2010). pp. 297-300, 2010.

[Z18] Zaidenberg N. J. "Hardware Rooted Security in Industry 4.0 Systems", in *Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures*, 51: 135-151, 2018.

[ZBYL20] Zaidenberg N.J., Ben Yehuda R. and Leon R., "ARM HYPERVISOR and TRUSTZONE ALTERNATIVES", accepted in *Encyclopedia of Criminal Activities and the Deep Web*, 2020.

[ZKBY+] Zaidenberg N.J., Kiperberg M., Ben Yehuda R., Leon R., Algawi A., and Resh A., "Memory introspection and hypervisor-based malware honeypot", *ICISSP Conference*, Submitted.

[ZNK+15] Zaidenberg N.J., Neittaanmäki P., Kiperberg M., Resh A., "Trusted Computing and DRM", in *Cyber Security: Analytics, Technology and Automation*, 2015.

[ZR15] Zaidenberg N.J. and Resh A., "Timing and side channel attacks", in *Cyber Security: Analytics, Technology and Automation*, 2015.

[ZWZ+10] Zhang L., Wang L., Zhang R., Zhang S., and Zhou Y., "Live memory acquisition through firewire", in *International Conference on Forensics in Telecommunications, Information, and Multimedia*, pp. 159-167, 2010.

ORIGINAL PAPERS

PI

HYPERVISOR-BASED WHITE LISTING OF EXECUTABLES

by

R. Leon, M. Kiperberg, A.A. Leon Zabag, A. Resh, A. Algawi, N.J. Zaidenberg
2019

IEEE Security & Privacy, Vol. 11, No. 5., pp. 58-67

https://doi.org/10.1109/MSEC.2019.2910218

Hypervisor-based Whitelisting of Executables

Roee S. Leon*[1], Michael Kiperberg[2], Anat Anatey Leon Zabag[2], Amit Resh[3], Asaf Algawi[1], and Nezer J. Zaidenberg[4]

[1] Department of Mathematical IT
University of Jyväskylä
Finland
Emails: roee.leonn@gmail.com, asaf.algawi@gmail.com
[2] Faculty of Sciences
Holon Institute of Technology
Israel
Emails: michaelkip@hit.ac.il, anatey.zabag@gmail.com
[3] School of Software Engineering
Shenkar College of Engineering, Design and Art
Israel
Email: amitr44@gmail.com
[4] School of Computer Science
The College of Management Studies
Israel
Email: nzaidenberg@me.com

***Abstract*—This paper describes an efficient system for ensuring code integrity of an OS, including its own-code and applications. We claim that the proposed system can protect from an attacker that has full control over the OS kernel. An evaluation of the system's performance suggests that the induced overhead is negligible.**

***Index Terms*—Security, application whitelisting, authorized execution, virtual machine monitors, secure boot, hypervisors**

I. Introduction

The problem of unauthorized software execution is well known. Malicious programs can corrupt or steal sensitive user data or sabotage the normal course of execution. Current methods of preventing unauthorized execution can be divided into three categories [1]:

1) Behavioral: these systems analyze the behavior (e.g., network or hard-drive activity) of the system and compare the current with a predefined behavior. If these behaviors differ, the environment is considered breached.
2) Signature-oriented: these systems contain a database of code samples that are known to be malicious. Every loaded executable is scanned and if it contains a code sample that is present in the database, then the environment is considered breached. Most anti-virus programs can be categorized as signature-oriented.
3) Whitelist-oriented: these systems contain a database of allowed executables. The criteria used for whitelisting is frequently based on one or more file attributes (e.g., file-path or cryptographic hash) [2]. Unlike signature-based systems, only these executables are allowed to run. These systems typically intercept every loaded executable and check whether it is contained within the database. If not, then the environment is considered breached.

The strength of behavioral systems is difficult to evaluate because these systems are based on heuristics[4]. Signature-based systems can protect only against attacks that were previously discovered and analyzed, and are, therefore, ineffective against zero-day attacks [3]. Whitelist-based systems provide the strongest protection guarantees [5] but are also the most restricting. For example, in order to install a new program, the system administrator must allow this program to be installed by inserting it into the whitelist database. Typically, whitelist enforcement is performed by intercepting executable images at their load time (e.g., by intercepting system-calls) [6]. In the event that there is a vulnerability, exploitation becomes possible in runtime [7][8]. Nonetheless, in environments that do not tend to change frequently, the preferred option is a whitelist-based system.

Protection systems differ not only in their modus operandi, but also in their mechanisms for self-protection. The system must protect its whitelist database and also itself. Some systems use agent-network verifiers that periodically checksum different portions of the system [9]. Others store their critical code in kernel mode (the OS privilege-level), assuming, reasonably, that the OS is less vulnerable to attacks than regular programs [10]. Due to their relatively large attack surface, OSes with monolithic/hybrid kernels, such as Linux and Windows, require additional protection mechanisms [9].

Our method can be categorized as whitelist-based as it permits the creation of a whitelist database of allowed executables that will be used by the system to enforce authorized execution. However, our method does not suffer from two main deficiencies present in current methods:

1) In our method, the execution of a given executable image (both in user mode and kernel mode) is enforced during its entire lifetime.
2) In our method, the system can prevent execution of unauthorized code even in case an attacker has full control over the OS kernel.

We consider an attacker that has (1) remote access to the machine and (2) full control over the OS kernel and peripherals. In addition, we assume that the UEFI firmware is trusted. We argue that given the described attacker and the given assumption, the described system can withstand (1) malicious code execution in user mode or kernel mode, and (2) attacks that involve malicious *DMA* memory writes using peripherals.

To provide such strong security guarantees, our system uses a hypervisor. We show that the performance degradation of the proposed system is negligible.

A hypervisor is a software module that is able to monitor and control the execution of an OS. These capabilities are provided by an extension to the original processor's instruction set, called "virtualization extensions". Virtualization extensions are available on processors designed by Intel (VT-x) [11], AMD (AMD-V), and ARM (Virtualization Extensions). Our method is implemented on Intel processors but can be easily ported to AMD and ARM. In section *VI* we discuss how our method can be ported to the *ARM* architecture.

Throughout this paper, we refer to the entity that wants to protect the system as the *system administrator*.

A. *VMX*

Many modern processors are equipped with a set of extensions to their basic instruction set architecture that enables them to execute multiple OSes simultaneously. This paper discusses Intel's implementation of these extensions, which they call Virtual Machine Extensions (VMX). The software that governs the execution of the operating systems is called a "hypervisor" and each OS (with the processes it executes) is called a "guest". Transitions from the hypervisor to the guest are called "vm-entries" and transitions from the guest to the hypervisor are called "vm-exits". While vm-entries occur voluntarily by the hypervisor, vm-exits are caused by some event that occurs during the guest's execution. The events may be synchronous, e.g., execution of an INVLPG instruction, or asynchronous, e.g., page-fault or general-protection exception. The event that causes a vm-exit is recorded for future use by the hypervisor. A special data structure called the Virtual Machine Control Structure (VMCS) allows the hypervisor to specify the events that should trigger a vm-exit as well as many other settings of the guest.

Intel's Extended Page Table (EPT), a technology generally called Secondary Level Address Translation (SLAT) allows the hypervisor to configure a mapping between the physical address space, (as it is perceived by a guest) and the real physical address space. Similarly to the virtual page table, EPT allows the hypervisor to specify the access rights for each guest's physical page. When a guest attempts to access a page that is either not mapped or has inappropriate access rights, an event called EPT-violation occurs, triggering a vm-exit.

Input-Output Memory Management Unit (IOMMU) specifies the mapping of the physical address space as perceived by the hardware devices to the real physical address space. It is a complementary technology to the EPT that allows the hypervisor to construct a coherent guest physical address space for both the OS and the devices.

B. *System description*

The system described in this paper consists of a UEFI [12] application and an executable scanner. The executable scanner creates a whitelist database that stores hashes of executable images' pages within an initially trusted system. The UEFI application initializes a hypervisor that monitors the execution of the system by running the OS as a guest. Whenever the guest attempts to execute a page that was not previously approved, a vm-exit occurs. The hypervisor computes the hash of the page to be executed and compares it against the appropriate record in the database. If a match is found, then the page is given execution rights and the hypervisor performs a vm-entry to continue the normal execution of the system.

We use the UEFI secure boot feature to guarantee the integrity of the UEFI application before it is executed by the UEFI firmware. The UEFI application reads the whitelist database from the disk into the main memory and then initializes a hypervisor. The hypervisor configures the EPT and the IOMMU such that the whitelist database and the hypervisor's code and data are not accessible either from the guest or from a hardware device.

II. Preparations

The system administrator needs to scan an initially trusted system and install the necessary files on a target machine. Afterwards, he needs to configure secure boot. These processes are described in the following paragraphs.

A. *User mode scanning*

The executable scanner runs on an initially trusted system. It recursively looks for all executable images; specifically, executables and shared-objects. In x86-64, memory accesses are RIP-relative. That is, the access offset to local symbols can be computed in advance by the static linker. Therefore, modifications to code that reference local symbols will not be needed in runtime.

An executable image may have many runtime dependencies. The runtime dependencies of an executable image are handled by the dynamic linker. Fortunately, in Linux, the dynamic linker performs modifications only to the data segment of the executable image. Therefore, the executable scanner simply hashes the executable segment of every executable/shared-object, in a page granularity, and stores the results consecutively in the database. Fig. 1 depicts the process. After all executable images are scanned, the executable scanner lexicographically sorts the hashes.

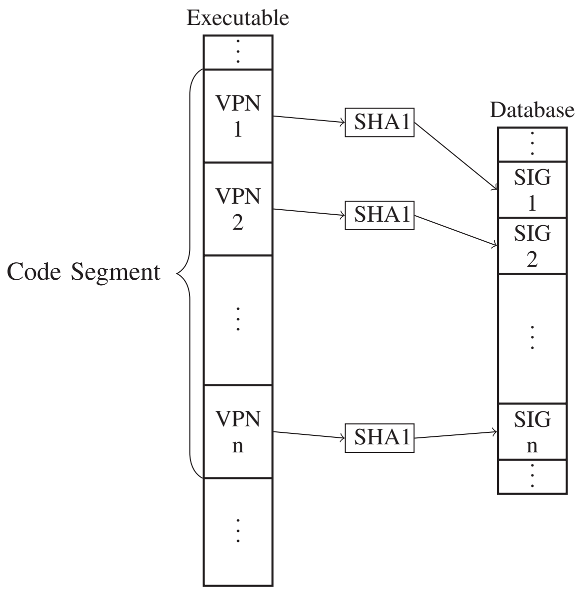


Fig. 1. The user mode executable image to be signed (left) is composed of one code segment, divided to virtual-pages. Each page is signed using SHA1 and the result is stored in the whitelist database (right).

B. Kernel mode scanning

The Linux kernel is composed of a statically linked executable image (*vmlinux*) and potentially loaded kernel modules. Kernel modules are different in their nature of execution than executable and shared-object files. This difference is reflected in two ways:

- Kernel modules are linked into the running kernel upon loading. That is, resolving of internal/external used symbols is undertaken solely by the kernel. These modifications vary between resets as kernel modules are not loaded to fixed addresses.
- The Linux kernel performs various static/dynamic modifications to the loaded code. For example, when compiled with ftrace, the first 5 bytes of each function are reserved for the Linux kernel internal tracer. These bytes are patched to no-ops at load time. Other possible modifications, that can be disabled/enabled in compilation time, are jump-labels, paravirt-operations (x86 specific) and alternatives. Fortunately, most of these modifications take place only at load time.

The user mode executable images scanning process has many advantages: (a) a simple whitelist database structure, (b) reduced runtime monitoring complexity, and (c) reduced runtime monitoring performance overhead. The whitelist database is simply composed of lexicographically sorted hashes. The latter allows the hypervisor to quickly look for a match. In addition, its code can be kept to a minimum. Similarly to user mode, the Linux kernel, by default, aligns code sections on page boundaries to ensure complete separation of the code and the data. In addition, as explained, most of the modifications take place only at load time. To retain the executable scanner user mode modus-operandi for kernel modules, we have performed the following steps:

1) During its initialization, the Linux kernel may optionally mount an initial ramdisk (*initramfs*), if found. The purpose of the initial ramdisk is to mount the root file system. In many Linux distributions (mainly general distributions), the initramfs also includes kernel modules because the machine on which the kernel will run is not known in advance. We built the latest stable Linux kernel to date (4.15.10, 19.03.2018) on a random machine, having all the necessary kernel modules statically compiled into the kernel. The latter can be achieved using 'localyesconfig' make target. Because the necessary kernel modules are statically compiled into the kernel, it was not necessary to boot the system with an initial ramdisk.
2) We booted up the system (Ubuntu 16.04.4 LTS) with the just-built kernel and disabled kaslr. Afterwards, we scanned the kernel directly from the main memory. Because the executable scanner cannot directly access kernel space memory (as it executes in user mode), the hypervisor provides a hypercall service that can be used to compute a hash of a given kernel page. The executable scanner uses the latter service to generate hashes of all active kernel-pages. These hashes are written directly into the whitelist database consecutively (just as in user mode applications).

It is worth noting that all the information regarding possible kernel-code modifications, both in kernel modules and the kernel image, is located within the corresponding images. In addition, the initial ramdisk can be mounted and scanned by the executable scanner. However, we chose to omit these capabilities from the executable scanner and the hypervisor due to the induced overhead and complexity.

C. Configuring secure boot

"Secure boot" is a feature provided by UEFI that allows a computer system owner to authenticate UEFI applications prior to their execution, thereby protecting the executable image from malicious modifications.

The UEFI specification defines four non-volatile variables used to control secure boot:

- platform key (*pk*)
- key exchange key (*kek*)
- signature database (*db*)
- forbidden signature database (*dbx*)

The most prominent variables are the platform key and the key exchange key. The platform key can contain one entry at most; typically, an x509 public key that belongs to the hardware vendor. The platform key can be used to sign *kek* keys. The *kek* variable may contain more than one entry. Each of the *kek* keys can be used to sign trusted executable images. Typically, the *kek* variable contains one or more keys that belong to the OS vendor. The *db* variable holds a whitelist database of executable images while the *dbx* variable holds a blacklist database of executable images. Updates to the *db* and *dbx* variables need to be signed using one of the keys within the *kek* variable.

Obviously, without knowing the private keys of the OS vendor, it is not possible to manipulate the secure boot variables. However, it is possible to rewrite all the keys. This process is referred to as taking control over the platform. Alternatively, it is possible to use an application called *SHIM*, which is signed by Microsoft. *SHIM* validates and loads another application. The validation is performed against a special boot service only (i.e., can be manipulated only during boot) UEFI variable, *MokList*. Unlike the secure boot variables, *MokList* can be modified without providing the private key of the OS vendor. Typically, *SHIM* launches a *MokList* management application that allows modifying the *MokList* variable in case the boot validation process failed. At this point, it is possible to add the signature of the desired UEFI application to the *MokList*.

To utilize secure boot, for the sake of our UEFI application verification, the system administrator has two options. Steps for option 1:

1) Reset the platform key. This can be done by entering UEFI setup mode.
2) Create key-pairs for *KEK*, *DB*, and *PK*.
3) Write the just-created keys to the corresponding UEFI variables in the specified order.
4) Sign our UEFI application using the created *KEK* or *DB* keys.
5) Copy the resultant signed UEFI application to the ESP partition.
6) Reboot the system.

Steps for option 2:

1) Copy the *SHIM* and the *MokList* management applications into the ESP partition.
2) Reboot the system.
3) Add our UEFI application signature to the *MokList* using the *MokList* management application.
4) Reboot the system.

D. Target installation

When the system's boot mode is configured to UEFI after a successfull startup, the UEFI boot manager loads a sequence of executable images, called UEFI applications. The UEFI firmware stores the location at which these images reside in a non-volatile storage. The boot-sequence can be configured using the firmware setup screen. The UEFI boot manager loads an executable image into the main memory, undertakes the necessary fixups, and executes its main routine. In case the entry routine returns, the UEFI boot manager proceeds to the next executable image, if there is one. The UEFI application's entry routine may also not return. A typical example for the latter is an OS loader implemented as a UEFI application.

The system described in this paper is implemented as a UEFI application. A system administrator interested in installing the system needs to perform the following steps:

1) Configure secure boot (as described in the previous subsection)
2) Install the UEFI application into a location accessible by the firmware. (e.g., a USB stick or a TFTP server.)
3) Install the whitelist database file into a storage device that is accessible by the firmware. For example, we recommend it is placed within the ESP partition on which the UEFI application resides.

III. Operation

The UEFI application, during its execution, obtains the whitelist database file from the disk, initializes a hypervisor, and returns to UEFI firmware. The UEFI firmware then proceeds to the next boot option which is typically the OS bootloader. The hypervisor remains in the main memory and continues its operation even after the application terminates. The hypervisor is set to detect code execution attempts both in user mode and kernel mode. When such an attempt is detected, the hypervisor verifies the page to be executed using its whitelist database. In the case of a valid hash, the hypervisor resumes the execution of the guest. The rest of this section provides a detailed explanation about the initialization and the operation of the system.

A. Initialization

The UEFI application starts by allocating a persistent memory block (i.e., the memory block can be used even after the application terminates) using UEFI boot services. The UEFI application loads the whitelist database into the allocated memory block using UEFI's file I/O services.

Afterwards, the UEFI application verifies the authenticity of the whitelist database using our built-in hardcoded certificate. Next, the UEFI application allocates another persistent memory block and initializes a hypervisor. During the hypervisor initialization, the EPT and the IOMMU are set up. Both the EPT and the IOMMU define not only the mapping of the perceived page but also its access rights. The hypervisor sets the EPT and IOMMU mappings by performing the following steps:

1) The hypervisor sets an identity mapping between the real physical address space and the guest physical address space. The latter is done by configuring the EPT such that guest physical page *X* translates to host physical page *X*. Fortunately, setting up identity mapping between the real physical address space and the I/O peripherals physical address space is trivial as the page-table hierarchy used by the EPT can also be used by the IOMMU.
2) The hypervisor sets the access rights of the hypervisor's code and data to read-only. This step ensures that malicious code, even if it executes in kernel mode, cannot modify the hypervisor's code and data.
3) The hypervisor sets the access rights of the remaining physical address space to write-only. This step ensures that any execution attempt will trigger a vm-exit, thus allowing the hypervisor to validate the faulting page.

Fig. 2 depicts the physical address space as it is perceived by the guest and I/O peripherals.

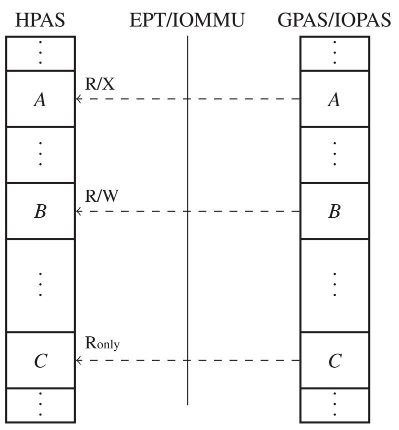


Fig. 2. Physical address space as perceived by the guest and I/O peripherals (right). Physical page *A* contains code that was previously authenticated by the hypervisor. Therefore, it has read/execute rights. Physical page *B* has yet to be executed. Therefore, it has read/write rights. Physical page *C* contains the hypervisor's code/data. Therefore, it has read only rights.

B. System monitoring

The hypervisor waits for an EPT violation to occur. When a vm-exit occurs, the processor saves the guest's state to VMCS, loads the hypervisor's state from VMCS, and begins execution of the hypervisor's predefined vm-exit handler. The handler checks whether the vm-exit was due to an EPT violation. Among the information stored in VMCS are the EPT violation reason and the guest physical address that caused the EPT violation. Due to the nature of our method, a page can either be executable or writable, but not both. Therefore, all the EPT violations are attributed to an attempt to write or to execute.

- If the violation was due to a write attempt, the hypervisor then removes the execute rights from the violating page, grants it write rights, and performs a vm-entry.
- If the violation was due to an execution attempt, the hypervisor then computes the hash of the violating physical page and looks for the resultant hash in its whitelist database. If a match is found, the hypervisor then removes the write rights from the violating page, grants it execute rights, and performs a vm-entry. If a match is not found, in case the violation occurred in user mode, the hypervisor injects a general-protection fault to the guest OS. Otherwise, if the violation occurred in kernel mode, the hypervisor then freezes up the system. Typically, the OS reacts to general-protection in user mode by stopping the running process.

C. OS kernel monitoring

When it comes to kernel mode, enforcing an unauthorized execution cannot always be done lazily (i.e., only at the time of a violation). In kernel mode, some actions are time critical. For example, acknowledging an interrupt to the PIC cannot cause an EPT violation as interrupt requests of equal or lower priority will not be generated until the page is given execution rights and an acknowledgement is sent to the PIC. Recall that the hypervisor initializes the EPT such that the entire guest physical address space has write-only access rights. That is to say, potentially time-critical kernel code will trigger a vm-exit due to an EPT violation upon execution attempt. To overcome this issue, the hypervisor verifies the kernel code pages and grants these pages execution rights.

Because we compiled the needed kernel modules statically into the kernel, there should be no more EPT violations due to kernel execution attempts. Nevertheless, if such an attempt is encountered, the hypervisor simply freezes up the machine.

IV. Security

We consider an attacker that has (1) remote access to the machine and (2) full control over the OS kernel and peripherals. In addition, we assume that the UEFI firmware is trusted. We argue that given the described attacker and the given assumption, the described system can withstand (1) malicious code execution in user mode or kernel mode, and (2) attacks that involve malicious *DMA* memory writes using peripherals.

In this paper, we assume that the UEFI firmware is trusted. This assumption can be relaxed by integrating a hardware root of trust method into our system. An Example of such method is the Intel Boot Guard technology, which allows verification of the boot process by flashing a public key into an OTP memory. In this way, the firmware code is verified on each subsequent boot. Obviously, once enabled, Intel Boot Guard cannot be disabled. We argue that the described system will prevent any unknown malicious code in user mode or kernel mode from executing.

Our method, being a whitelist system, prevents execution of unauthorized code. However, attacks in which the attacker manipulates the control flow of a program (e.g., by causing a return instruction to pass control to an existing code of his choosing) are possible. In Section *VI* we discuss how our system can be further extended to provide protection from such attacks.

A. HV memory protection

Secondary Level Address Translation (SLAT) is a mechanism implemented as part of hardware-assisted virtualization technology to reduce the overhead of managing the hypervisor's guest page-tables. SLAT is supported by Intel (EPT), AMD (RVI), and ARM (Stage-2 page-tables). Simply put, SLAT allows the hypervisor to control the mapping of physical pages addresses as they are perceived by the guest (known as guest-physical-address) to real physical pages addresses (known as host-physical-address). An analogy to SLAT usage in a virtualized environment (i.e., controlled by a hypervisor), is virtual page-tables usage in a process context in a non-virtualized environment (i.e., controlled by an OS). Fig. 3 depicts the guest's address translation process.

The Input Output Memory Management Unit (IOMMU) is a memory management unit that stands between DMA-capable peripherals and the main memory. In this sense, it functions as a virtual page-table for devices. DMA is a

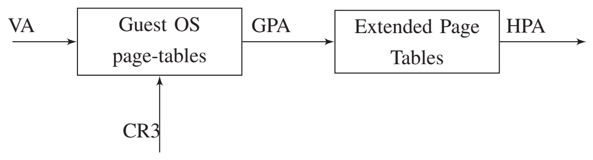


Fig. 3. Address translation process of a guest in Intel processors. First, the guest-linear address is translated into a guest-physical-address by the OS page-tables. Second, the guest-physical-address is translated into a host-physical-address by the hypervisor's extended-page-tables

hardware mechanism that allows peripherals to access the main memory directly without going through the processor. The IOMMU allows the OS or hypervisor to set paging-structures for the peripherals. That is, the peripherals will access a virtual-address (also known as I/O address) that will be translated by the IOMMU.

To protect itself against malicious modifications, the hypervisor configures both the EPT and the IOMMU in such a way that all of its sensitive memory regions are not mapped and, therefore, are not accessible either from the guest or from a hardware device.

B. *Secure boot*

"Secure boot" is a feature provided by UEFI that allows a computer system owner to authenticate UEFI applications prior to their execution, thereby protecting the executable image from malicious modifications.

The first phase in the UEFI boot process is the firmware initialization phase. The firmware initialization phase is also called the security (SEC) phase because it serves as the basis for the root of trust. After the completion of the SEC phase, the trust is maintained via public key cryptography.

The UEFI secure boot feature is essential for the security of our system. Consider, for example, an attacker that has a remote root access to a machine. In addition, the media that contains the UEFI application is plugged into the computer. The attacker can mount the partition on which the UEFI application resides and modify the executable image as required. As a result, during the next boot, the UEFI firmware will load the malicious executable image.

V. Performance

The proposed system goes into action when an EPT violation occurs. Recall that due to the nature of our method, a page can either be executable or writable but not both. Therefore, all EPT violations are due to an attempt to write or to execute. Whenever a page requires execution rights, the hypervisor computes its hash and searches for a match within the whitelist database. When a page requests write rights, the hypervisor simply removes its execution rights and grants it write rights instead.

In the first experiment, we tried to estimate the induced overhead due to the aforementioned by forcing the OS to page-out a code page every time it is accessed. Therefore, it has to be brought up from the disk and written to memory before it can be executed. The results show that the extra overhead is negligible compared with the time it takes to read the page from the disk and write it to memory.

In the second experiment, we performed an empirical evaluation of the system. We picked an open-source benchmarking software and ran several types of benchmarks to assess the impact of our system on a randomly selected computer. The results show that the induced overhead is negligible.

All experiments were performed in the following environment:

- CPU: Intel Core i5-4570 CPU @ 3.20GHz (4 physical cores - only 1 core was enabled)
- RAM: 8GB
- OS: Ubuntu 16.04.4 LTS - customized kernel 4.15.10 as described in section II.

A. *Page verification forcing experiment*

In this experiment, we took a large executable file (10MB) and modified one of its code pages such that the first byte of the page was 0xc3 (return-from-procedure opcode in x86). We wrote an application that requests a mapping of the aforementioned file into its virtual address space with full access rights (read, write, and execute). Next, using the *fadvise64* system call, we instructed the OS not to keep the file in memory. The size of the file, along with the advise caused the access to any page within the mapped file to always generate a major page fault (i.e., it forced the OS to access the disk). Then, using *rdtsc*, we measured the number of cycles it takes to perform the call to our modified page and return, with and without active page verification. We ran the application a total of 100,000 times. As can be seen by the results presented in Fig. 4, the performance penalty of the active page verification is less than one percent.

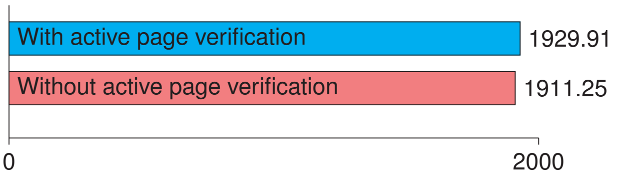


Fig. 4. Thousands of cycles (less is better) for a single call and return.

B. *Empirical evaluation*

In this experiment, we tested the system in three scenarios:

- without hypervisor
- with hypervisor and disabled page verification
- with hypervisor and enabled page verification

We selected an open-source benchmarking software, Phoronix Test Suite [13] (PTS) v5.2.1 (Khanino). Six tests were conducted:

(a) unpack-linux: Linux kernel unpacking, disk-intensive, default configuration.
(b) compress-7zip: 7-Zip compression test, cpu-intensive, default configuration.

(c) *dbench-6client*: Dbench disk performance test, disk-intensive, 6-client configuration.
(d) *dbench-48client*: Dbench disk performance test, disk-intensive, 48-client configuration.
(e) *ramspeed*: System memory performance test, memory-intensive, copy and integer configuration.
(f) *git*: Sample git operations, general system benchmark, default configuration.

As can be seen in the results reported in Fig. 5, the performance penalty of the hypervisor is no more than 5% compared with No-HV, whereas compared with the performance penalty of the HV with page verification, it is no more than 2% compared with HV only.

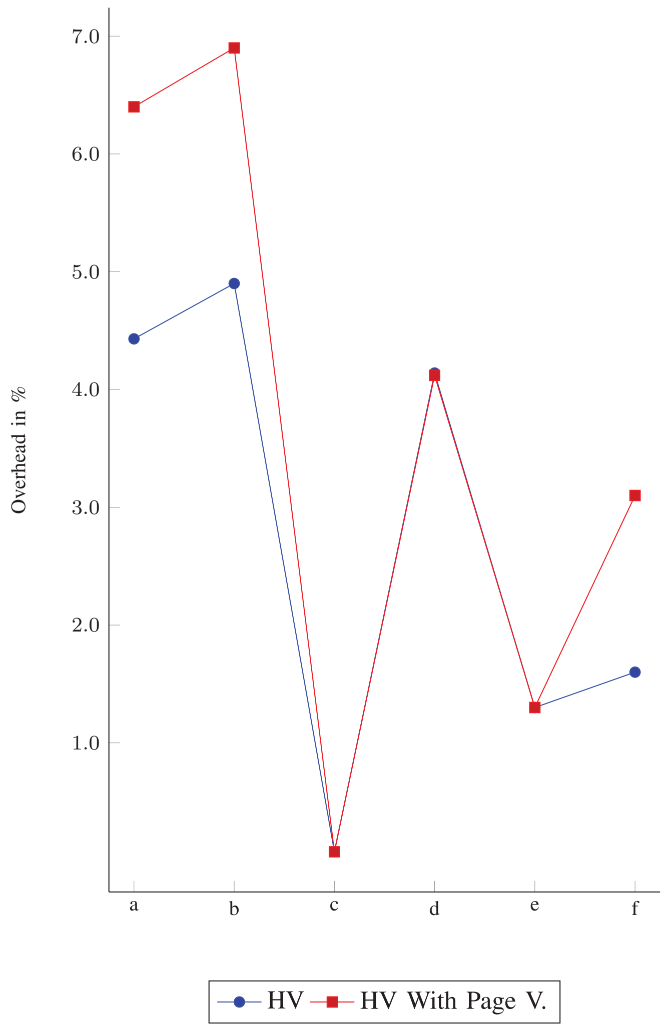


Fig. 5. Overhead of the benchmark execution under two conditions: (a) with HV only, and (b) with HV and enabled page verification

VI. Limitations and Future Work

Our method suffers from several limitations. These limitations and possible solutions are described in the next paragraphs.

A. SMP support

Our method currently supports only one processor. Recall that whenever an EPT violation occurs, the processor needs to update the EPT structures with the correct access rights (write or execute). The processor may cache information from the EPT paging structures. That is, in a multi-processor system, the processor that caused the violation will have to gather all the processors to make sure that their internal caches are flushed after setting up the new rights. This process is very common and is usually referred to as TLB shootdown. Due to the relatively high turnover of user mode pages, this gathering process can induce significant overhead.

A possible optimization to the aforementioned performance problem is based on the fact that it is not always necessary for all processors to have an identical EPT paging structure at any given point in time as we do not modify the actual mappings but only the access rights. For example, if processor A needs to set execution rights for a physical page x and processor B has only read rights for physical page x, then processor A can freely modify its EPT paging structure without gathering processor B.

B. Other OSes support

Supporting other OSes is indeed possible as we do not perform any modifications whatsoever to the running kernel. However, other OSes may behave differently, both in kernel mode and user mode. For example, Windows may modify the program's code at load time. These modifications, however, are not difficult to handle because all information about them is located within the PE file. In addition, they all take place only at load time. Examples of such modifications are relocations and security cookies that if they exist, are stored within the PE executable file. The former is stored within a special section while the latter is stored in a PE data-directory.

Despite the security consequences of having both code and writable data on the same page (for example, this arrangement breaks DEP), there are still OSes on which it is possible. The latter may introduce two problems to our current method: (a) a partial code page, (b) a self-modifying page. If the data part of a partial code page is modified during runtime, then its hash might not match. Fig. 6 illustrates the problem.

Consider the same scenario as described in the previous problem, but this time page A modifies its own data. As a result, the system will enter an infinite write/execute EPT violation loop.

The second problem becomes like the first problem by emulating the write operation. However, the bigger challenge is to decide whether the written data is legitimate or malicious. We argue that the executable scanner can be modified to support legitimate runtime modifications.

C. Managed code

Our method is very efficient and effective when execution of native-code is considered because it is executed directly by the processor. On the other hand, managed and interpreted code is typically executed by another application usually referred

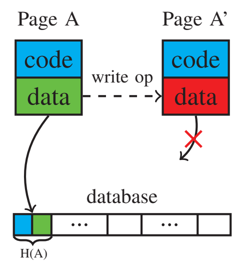


Fig. 6. Page *A* is a partial code page (i.e., contains both code and data). Initially, the data part has not yet been modified; therefore, *H(A)* is located within the whitelist database. Later, the data part of page *A* is modified. Even though the code is left untouched, an execution attempt of *A'* will fail as *H(A')* is not located in the whitelist database.

to as a virtual machine. If the virtual machine itself is signed, then the hypervisor will allow it to freely execute possibly malicious code.

A possible solution to managed and interpreted code is to to store the hashes of the managed and interpreted code within the whitelist database. The hypervisor can intercept the virtual machine application attempts to load the code and perform a hash validation. The hypervisor can possibly detect such attempts by intercepting system-calls within the guest OS. This solution, however, will not handle cases in which the interpreted code is compiled into native code (i.e., JIT code).

D. *Control flow and data integrity*

Our method guarantees that no malicious modifications will be undertaken to executing code (both in user mode and kernel mode). However, our method does not provide user mode and kernel mode control flow and data integrity. For example, an attacker may modify the contents of the .got section, thus affecting the control flow of a program. Thankfully, control flow integrity is a heavily researched subject and many effective solutions exist[14][15]. We believe that such attacks can be mitigated by combining our method with a method that guarantees control flow integrity. Moreover, our system can be used to enforce authorized code execution on the used method.

E. *ARM architecture*

Our method can be ported to the ARM architecture on ARM devices (e.g., most of today's smartphones) that implement the virtualization extensions. ARM virtualization extensions provide capabilities similar to Intel VT-x. For example, they provide a mechanism similar to Intel EPT for guest-physical-address (IPA in ARM terminology) to host-physical-address translation. The ARM Security Extensions, known as TrustZone, provide a way to create an isolated environment in which sensitive applications can execute. This isolated environment executes at the highest privilege level (higher than the hypervisor) and is not subject to virtualization. Due to the latter, for better security guarantees, our system might use TrustZone in addition to a hypervisor.

F. *Other applications*

Our method can be further extended to provide other useful security applications. An example of such application is a sandbox for runtime analysis of malware. This can be done by entering a special monitor mode in case of an execution violation. In monitor mode, the behavior of the violating process may be inspected. Examples of potentially interesting behaviors are system-calls initiated by the process and code executed by the process.

VII. Conclusions

We have seen that current whitelist-based systems have deficiencies that make them impractical, particularly in the case of code modification attacks. We have described a system that will prevent any unauthorized native-code from being executed. We explained in detail how the described system can be installed and even verified on each subsequent boot. We also showed that the performance overhead of the proposed system is negligible. The described system has a few limitations. However, as described, most of these limitations can be overcome without much effort. The described system can be further extended to provide other useful security applications. We believe that in addition to VMX, the Intel SGX can be used to provide data integrity for user mode applications.

References

[1] Idika, Nwokedi, and Aditya P. Mathur, "A survey of malware detection techniques," Purdue University 48 (2007).
[2] H. Pareek, S. Romana and P. R. L. Eswari, "Application whitelisting: approaches and challenges," International Journal of Computer Science, Engineering and Information Technology (IJCSEIT), Vol.2, No.5, 2012.
[3] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa, "Fast Signature Generation for Zero-day PolymorphicWorms with Provable Attack Resilience," IEEE Symp. Security and Privacy, 2006.
[4] K.M.C. Tan, K.S. Killourhy and R.A. Maxion, "Undermining an anomaly-based intrusion detection system using common exploits," RAID, 2002.
[5] Steve Mansfield-Devine, "The promise of whitelisting," Network Security Volume 2009, Issue 7, July 2009, Pages 4-6
[6] Fanton et al., "Secure System For Allowing The Execution of Authorized Computer Program Code," U.S. Patent No. 9,665,708 B2, May 30, 2017
[7] Jim Beechey, "Application Whitelisting: Panacea or Propaganda," https://www.sans.org/reading-room/whitepapers/application/application-whitelisting-panacea-propaganda-33599 [Online; accessed 26-March-2018].
[8] NCC Group Publication, "Bypassing Windows AppLocker using a Time of Check of Use vulnerability," December 2013.
[9] Aumaitre, Damien, and Christophe Devine, "Subverting windows 7 x64 kernel with dma attacks," HITBSecConf Amsterdam (2010).
[10] Munir Kotadia, "Norton AntiVirus ignores malicious WMI instructions," CBS Interactive. October 21, 2004. Archived from the original on September 12, 2009. Retrieved April 5, 2009.
[11] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2007, vol. 3.
[12] UEFI, "Unified Extensible Firmware Interface (UEFI) Specification," August 2017
[13] M. Larabel and M. Tippett, "Phoronix test suite," http://www.phoronix-test-suite.com/ [Online; accessed 26-March-2018].
[14] V. Pappas, "kBouncer: Efficient and transparent ROP mitigation," Technical report, Columbia University, 2012
[15] Y. Cheng, Z. Zhou, Y. Miao, X. Ding and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," In Symposium on Network and Distributed System Security (NDSS), 2014

Roee Shimon Leon was born in Tel Aviv, Israel, in 1989. He received his B.Sc. (Cum Laude) in Software-Engineering from Shenkar College of Engineering and Design, Israel, and M.Sc from the University of Jyväskylä, Finland, both in 2015. As of 2019, Roee is a Ph.D. student at the University of Jyväskylä, under the supervision of Prof. Pekka Neittaanmaki and Dr. Nezer Zaidenberg. Roee's research focuses on applications of hardware virtualization in security.
Contact email: roee.leonn@gmail.com

Michael Kiperberg was born in Ukraine in 1987 and immigrated to Israel in 1997, where he received his B.Sc. and M.Sc. degrees from the Tel Aviv University in 2008 and 2012, respectively. In 2015 Michael completed his Ph.D. studies in the University of Jyväskylä, Finland, under the supervision of Prof. Pekka Neittaanmäki and Dr. Nezer Zaidenberg. The title of his Ph.D. dissertation was: "Preventing Reverse Engineering of Native and Managed Programs". Michael joined the Holon Institute of Technology in 2016, where he teaches theoretical and applied courses in computer sciences. Michael's research focuses on applications of hardware virtualization in security.
Contact email: michaelkip@hit.ac.il

Anat Anatey Leon Zabag was born in Tel Aviv, Israel, in 1989. She received her BSc. in Computer-Science from Holon Institute of Technology, in 2018. Anat has 4 years of professional experience in hi-tech startup companies on which she worked as a researcher and a software developer; and is currently working towards her M.Sc. Anat's research focuses on applications of hardware virtualization in security.
Contact email: anatey.zabag@gmail.com

Amit Resh was born in Haifa, Israel, in 1959. He received his B.Sc. in Computer-Engineering and MBA from the Technion, Israel Institute of Technology, in 1986 and 2001 respectively. In 2013 he received his MSc. from the University of Jyväskylä, Finland. In 2016 he received his PhD. from the University of Jyväskylä, Finland. He has more than 30 years of professional experience in hi-tech companies in Israel and the USA. He has previously worked as Program-Manager at Apple and as VP of R&D at Connect One, as well as other companies in the embedded-systems industry. Currently he is COO of TrulyProtect, a company developing trusted computing systems based on virtualization technology. He is also a part-time staff member at Shenkar College of Engineering and Design, Israel.
Contact email: amitr44@gmail.com

Asaf Algawi was born in Haifa, Israel in 1986. He received his B.Sc. in Information Systems Engineering from Ben-Gurion University of the Negev, Israel, in 2009. In 2015, he received his M.Sc from the University of Jyväskylä, Finland. Asaf has 6 years of professional experience in hi-tech military units in the IDF; these include 3 years as a system analyst and another 3 years leading a small team of .NET developers. As of 2019, he is working towards his Ph.D. at the University of Jyväskylä, Finland. Asaf's research focuses on applications of hardware virtualization in security.
Contact email: asaf.algawi@gmail.com

Nezer Jacob Zaidenberg was born in Tel Aviv, Israel, in 1979. Nezer hold a B.Sc in Computer Science and Statistics and Operations Research, M.Sc in Operations Research and MBA, all from Tel Aviv University, Israel. Nezer completed his Ph.D. in 2012 in the University of Jyväskylä. Nezer has past work experiene with NDS, IBM and others. Nezer is the CTO of TrulyProtect.
Contact email: nzaidenberg@me.com

PII

HYPERVISOR-ASSISTED ATOMIC MEMORY ACQUISITION IN MODERN SYSTEMS

by

M. Kiperberg, R. Leon, A. Resh, A. Algawi, N.J. Zaidenberg 2019

ICISSP Conference

https://doi.org/10.5220/0007566101550162

Hypervisor-assisted Atomic Memory Acquisition in Modern Systems

Michael Kiperberg[1], Roee Leon[2], Amit Resh[3], Asaf Algawi[2] and Nezer Zaidenberg[4]

[1]*Faculty of Sciences, Holon Institute of Technology, Israel*

[2]*Department of Mathematical IT, University of Jyväskylä, Finland*

[3]*School of Computer Engineering, Shenkar College of Engineering, Design and Art, Israel*

[4]*School of Computer Sciences, The College of Management, Academic Studies, Israel*

michaelkip@hit.ac.il, roee.leonn@gmail.com, amitr44@gmail.com, nzaidenberg@me.com

Keywords: Live Forensics, Memory Forensics, Memory Acquisition, Virtualization, Reliability, Atomicity, Integrity of a Memory Snapshot, Forensic Soundness.

Abstract: Reliable memory acquisition is essential to forensic analysis of a cyber-crime. Various methods of memory acquisition have been proposed, ranging from tools based on a dedicated hardware to software only solutions. Recently, a hypervisor-based method for memory acquisition was proposed (Qi et al., 2017; Martignoni et al., 2010). This method obtains a reliable (atomic) memory image of a running system. The method achieves this by making all memory pages non-writable until they are copied to the memory image, thus preventing uncontrolled modification of these pages. Unfortunately, the proposed method has two deficiencies: (1) the method does not support multiprocessing and (2) the method does not support modern operating systems featuring address space layout randomization (ASLR). We describe a hypervisor-based memory acquisition method that solves the two aforementioned deficiencies. We analyze the memory usage and performance of the proposed method.

1 INTRODUCTION

Nowadays, the sophistication level of cyber-attacks makes it almost impossible to analyze them statically. Many of the attacks are designed to detect debuggers and other tools of dynamic analysis. Upon detection of such tool, the malicious software deviates from its normal behavior, thus rendering the analysis useless. Therefore, usually the analysis of an attack is divided into two steps: memory acquisition and static analysis. In the first step, a software (Qi et al., 2017; Martignoni et al., 2010; Reina et al., 2012) or a hardware tool (Zhang et al., 2010) acquires the memory contents of a running system and stores it for later analysis. In the second step, a static analysis tool, e.g., Rekall (Cohen, 2014), is applied to the acquired image of memory to analyze the malicious software.

The memory acquisition is performed while the system is running and updating its data structures and pointers. Consider the following example. The operating system creates a new Process Environment Block at page 1,000 and adds it to the list of running processes at page 2,000. Assume that the process creating occurs when the first 1,500 pages were already acquired. In the resulting memory image, we will have a list of processes that point to an invalid Process Environment Block because page 1,000 was acquired before the creation of the Process Environment Block. Therefore, special measures must be taken to avoid inconsistencies in the acquired memory image.

This paper presents a software hypervisor-based tool for consistent memory acquisition. Hypervisor's ability to configure access rights of memory pages can be used to solve the problem of inconsistencies as follows:

1. When the hypervisor is requested to start memory acquisition, it configures all memory pages to be non-writable.
2. When an attempt is made to write to a memory page P, the hypervisor is notified.
3. The hypervisor copies the contents of P to its inner buffer and configures the page P to be writable.
4. The hypervisor periodically sends the data in its inner buffer. If more data can be sent than is available in the inner buffer, then the hypervisor sends other pages and configures them to be writable.

This method is described in multiple previous works (Qi et al., 2017; Martignoni et al., 2010).

Unfortunately, two problems arise with the described method in modern systems. The first problem is the availability of multiple processors. Each processor has direct access to the main memory and can freely modify any page. Therefore, when the hypervisor is requested to start memory acquisition, it must configure all memory pages *on all processors* to be non-writable.

Another problem is delay sensitivity of some memory pages. Generally, interrupt service routines react to interrupts in two steps: they register the occurrence of an interrupt and acknowledge the device that the interrupt was serviced. The acknowledgement must be received in a timely manner; therefore, the registration of an interrupt occurrence, which involves writing to a memory page, must not be intercepted by a hypervisor, i.e., these pages must remain writable.

Address space layout randomization, a security feature employed by modern operating systems, e.g., Windows 10, complicates the delay sensitivity problem even more. When ASLR is enabled, the operating system splits its virtual address space into regions. Then, during the initialization of the operating system, each region is assigned a random virtual address. With ASLR, the location of the delay-sensitive pages is not known in advance.

We propose the following solution to the problems mentioned above. Our hypervisor invokes an operating system's mechanism to perform an atomic access rights configuration on all the processors. Section 4.3 describes the invocation process, which allows our hypervisor to call an operating system's function in a safe and predictable manner.

We solve the delay sensitivity problem by copying the delay-sensitive pages to the hypervisor's inner buffer in advance, i.e., when the hypervisor is requested to start memory acquisition. The ASLR complication is addressed by inspecting the operating system's dynamic map of memory regions and obtaining the dynamic locations of the delay-sensitive pages. Section 4.2 contains a detailed description of ASLR as it is implemented in Windows 10 and our solution of the delay sensitivity problem.

The contribution of our work is:

1. We show how memory can be acquired on systems with multiple processors.
2. We present a solution to the delay sensitivity problem.
3. We explain how the locations of sensitive pages can be obtained dynamically on Windows 10. We believe that similar methods will be applicable to future versions of Windows.

2 RELATED WORK

Previous versions of the Windows operating system contained a special device, `\\Device\PhysicalMemory`, that mapped the entire physical memory. This device could be used to acquire the physical memory without any special tools. Unfortunately, because this method of memory acquisition relies on the operating system, a skilled attacker can disable or corrupt this feature (Carrier and Grand, 2004). Moreover, this method is not available in Windows 2003 SP1 and later versions of Windows (Microsoft Corporation, 2009).

Another method of memory acquisition is based on generic or dedicated hardware. Several previous works show how a generic FireWire card can be used to acquire memory remotely (Zhang et al., 2010). A dedicated PCI card, named Tribble, works in a similar manner (Carrier and Grand, 2004). The main advantage of a hardware solution is the ability of a PCI card to communicate with the memory controller directly, thus providing a reliable result even if the operating system itself was compromised. However, hardware solutions have three deficiencies:

1. They are expensive.
2. The produced memory image is not atomic.
3. These tools fail when Device Guard (Durve and Bouridane, 2017), a security feature introduced in Windows 10, is enabled.

Device Guard is a security feature that utilizes IOMMU (Ben-Yehuda et al., 2007; Zaidenberg, 2018) to prevent malicious access to memory from physical devices (Brendmo, 2017). When Device Guard is enabled, the operating system assigns each device a memory region that it is allowed to access. Any attempt to access memory outside this region is prevented by the DMA controller.

Recently, several hypervisor-based methods of memory acquisition have been proposed. HyperSleuth (Martignoni et al., 2010) is a driver with an embedded hypervisor. Its hypervisor is capable of performing atomic and *lazy* memory acquisition. The laziness is expressed in the ability of the hypervisor to continue the normal execution while the memory is acquired. ForenVisor (Qi et al., 2017) is a similar hypervisor with additional features that allow it to log keyboard strokes and hard-drive activity. Both hypervisors were tested on Windows XP SP3 with only one processor enabled.

We show how the idea of HyperSleuth and ForenVisor can be adapted to multi-processor systems executing Windows 10.

3 BACKGROUND

3.1 Hypervisors

The main component of the described system is a hypervisor, which utilizes the VMX instruction set extension. This section provides a short overview of this component. Section 4 contains a detailed description of the hypervisor's design There are two types of hypervisors: full hypervisors and thin hypervisors. Full hypervisors like Xen (Barham et al., 2003), VMware Workstation (VMware, 2018), and Oracle VirtualBox (Oracle, 2018) can execute several operating systems concurrently. The main goal of VMX was to provide software developers with means to construct efficient full hypervisors.

Thin hypervisors, in contrast, can execute only a single operating system. Their main purpose is to enrich the functionality of an operating system. The main benefit of a hypervisor over kernel modules (device drivers) is the hypervisor's ability to create an isolated environment, which is important in some cases. Thin hypervisors are used for operating system's integrity validation (Seshadri et al., 2007), remote attestation (Kiperberg et al., 2015; Kiperberg and Zaidenberg, 2013), malicious code execution prevention (Resh et al., 2017), in-memory secret protection (Resh and Zaidenberg, 2013), hard drive encryption (Shinagawa et al., 2009), and memory acquisition (Qi et al., 2017),

In general, because thin hypervisors are much smaller than full hypervisors, they are superior in their performance, security, and reliability. The hypervisor described in this paper is a thin hypervisor that is capable of acquiring a memory image of an executing system atomically. The hypervisor was written from scratch to achieve an optimal performance.

Similarly to an operating system, a hypervisor does not execute voluntarily but responds to events, e.g., execution of special instructions, generation of exceptions, access to memory locations, etc. The hypervisor can configure interception of (almost) each event. Interception of an event (a VM-exit) is similar to handling of an interrupt, i.e., a predefined function is executed by the processor. Another similarity with an operating system is the hypervisor's ability to configure the access rights to each memory page through a data structure, named EPT, which resembles the virtual page table. An attempt to write to a non-writable (according to EPT) page induces a VM-exit and allows the hypervisor to act.

3.2 Lazy Hypervisor-based Memory Acquisition

The solutions proposed by HyperSleuth and ForenVisor for memory acquisition are based on a thin hypervisor and can be summarized as follows. The hypervisor remains idle (or deactivated) until it receives a memory acquisition request. When the request is received, the hypervisor configures the EPT to make all memory pages non-writable. An attempt to write to a page P will trigger a VM-exit, thus allowing the hypervisor to react. The hypervisor reacts by copying P to an inner queue, and making P writable again. Future attempts to write to P will not trigger a VM-exit.

The queued pages are transmitted to a remote machine via a communication channel. This channel may be secure or not, depending on the security assumptions about the underlying environment. The size of the queue is dictated by the communication channel bandwidth and the volume of pages that are modified by the system. Obviously, if the communication channel allows sending more data than is available in the queue, then the hypervisor sends other non-writable pages and configures them to be writable. This process continues until all pages become writable.

3.3 Delay-sensitive Pages and ASLR

Generally, interrupt service routines react to interrupts in two steps: they register the occurrence of an interrupt and acknowledge the device that the interrupt was serviced. The acknowledgement must be received in a timely manner; therefore, the registration of an interrupt occurrence, which involves writing to a memory page, must not be intercepted by a hypervisor, i.e., these pages must remain writable. This issue was not addressed by the authors of HyperSleuth and ForenVisor. We assume that this problem did not occur on Windows XP SP3, which was tested in previous works.

Address space layout randomization, a security feature employed by modern operating system, e.g., Windows 10, complicates the delay sensitivity problem even more. When ASLR is enabled, the operating system splits its virtual address space into regions. Then, during the initialization of the operating system, each region is assigned a random virtual address. This behavior is useful against a wide range of attacks (Evtyushkin et al., 2016) because the location of potentially vulnerable modules is not known in advance. However, for the exact same reason, the location of the delay-sensitive pages is also unpredictable.

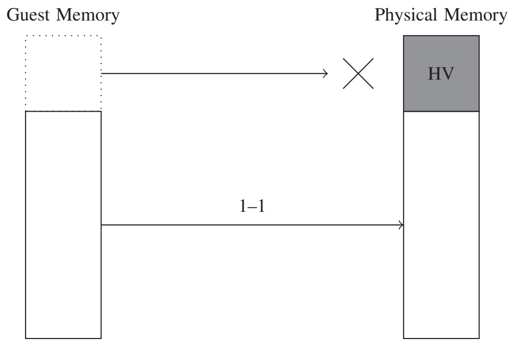


Figure 1: Mapping between the physical address space as observed by the operating system (left) and the actual physical address space. The mapping is an identity mapping with the exception of the hypervisor's pages, which are not mapped at all.

4 SYSTEM DESIGN

4.1 Initialization

Our hypervisor is implemented as a UEFI application (Unified EFI, Inc., 2006). The UEFI application loads before the operating system, allocates all the required memory, and initializes the hypervisor. After initialization, the UEFI application terminates, thus allowing the operating system boot loader to initialize the operating system. We note that while the application terminates, the hypervisor remains active.

In order to protect itself from a potentially malicious environment, the hypervisor configures the EPT such that any access to the code and the data of the hypervisor is prohibited. With this exception, the EPT is configured to be an identity mapping that allows full access to all the memory pages (Figure 1).

The hypervisor remains idle until an external event triggers its memory imaging functionality. The external event might be the reception of a network packet, insertion of a USB device, invocation of a system call, etc. In our prototype implementation, we used a special CPUID instruction, which we call `FREEZE`, as a trigger.

In response to `FREEZE`, the hypervisor performs two actions:

1. Locates and copies the delay-sensitive pages.
2. Requests all processors to configure the access rights of all memory pages to be non-writable.

When the configuration is complete, the hypervisor reacts to page modification attempts by making the page writable and copying it to an inner queue. The hypervisor exports the pages stored in the inner queue in response to another special CPUID instructions, which we call `DUMP`. If the queue is not full, then the hypervisor exports other non-writable pages and makes the exported pages writable.

Algorithm 1: Memory Acquisition.

```
1: file ← Open(...)
2: FREEZE()
3: while DUMP(addr, page) do
4:     Seek(file, addr)
5:     Write(file, page)
6: Close(file)
```

Algorithm 1 shows how `FREEZE` and `DUMP` can be used to acquire an atomic image of the memory. First, the algorithm opens a file that will contain the resulting memory image. Then, `FREEZE` is invoked, followed by a series of `DUMP`s. When the `DUMP` request returns *false*, the file is closed and the algorithm terminates.

4.2 Delay-sensitive Pages

Section 3.3 explains that certain pages must not be configured as non-writable. Moreover, due to ASLR, the hypervisor has to discover the location of these pages at run time based on the operating system data structures. This section presents the data structures of Windows 10 that can be used to locate the delay-sensitive pages.

Windows 10 defines a global variable `MiState` of type `MI_SYSTEM_INFORMATION`. The hypervisor can easily locate this variable as it has a constant offset from the system call service routine, whose address is stored in the LSTAR register (Table 1). The `MI_SYSTEM_INFORMATION` structure has a field named `Vs` of type `MI_VISIBLE_STATE`. Finally, the `MI_VISIBLE_STATE` structure has a field named `SystemVaRegions`, which is an array of 15 pairs. Each pair corresponds to a memory region whose address was chosen at random during the operating system's initialization. The first element of the pair is the random address and the second element is the region's size. A description of each memory region is given in Table 2. A more detailed discussion of the memory regions appears in (Russinovich et al., 2012). Our empirical study shows that that the following regions contain delay-sensitive pages:

1. MiVaProcessSpace
2. MiVaPagedPool
3. MiVaSpecialPoolPaged
4. MiVaSystemCache
5. MiVaSystemPtes
6. MiVaSessionGlobalSpace

Table 1: Windows ASLR-related Data Structures.

Offset	Field/Variable Name	Type
X	System Call Service Routine	*Code*
...	...	...
+0xFB100	MiState	MI_SYSTEM_INFORMATION
+0x1440	Vs	MI_VISIBLE_STATE
+0x0B50	SystemVaRegions	MI_SYSTEM_VA_ASSIGNMENT[14]
+0x0000	[0]	MI_SYSTEM_VA_ASSIGNMENT
+0x0000	BaseAddress	uint64_t
+0x0008	NumberOfBytes	uint64_t

Table 2: Memory Regions.

Index	Name
0	MiVaUnused
1	MiVaSessionSpace
2	MiVaProcessSpace
3	MiVaBootLoaded
4	MiVaPfnDatabase
5	MiVaNonPagedPool
6	MiVaPagedPool
7	MiVaSpecialPoolPaged
8	MiVaSystemCache
9	MiVaSystemPtes
10	MiVaHal
11	MiVaSessionGlobalSpace
12	MiVaDriverImages
13	MiVaSystemPtesLarge

Therefore, the hypervisor never makes these regions non-writable.

4.3 Multiprocessing

The hypervisor responds to `FREEZE`, a memory acquisition request, by copying the delay-sensitive pages to an inner queue and configuring all other pages to be non-writable. However, when multiple processors are active, the access rights configuration must be performed atomically on all processors.

Operating systems usually use inter-processor interrupts (IPIs) (Intel Corporation, 2018) for synchronization between processors. It seems tempting to use IPIs also in the hypervisor, i.e., the processor that received `FREEZE` can send IPIs to other processors, thus requesting them to configure the access rights appropriately. Unfortunately, this method requires the hypervisor to replace the operating system's interrupt-descriptors table (IDT) with the hypervisor's IDT. This approach has two deficiencies:

1. Kernel Patch Protection (KPP) (Field, 2006), a security feature introduced by Microsoft in Windows 2003, performs a periodic validation of critical kernel structures in order to prevent their illegal modification. Therefore, replacing the IDT requires also intercepting KPP's validation attempts, which can degrade the overall system performance.
2. Intel processors assign priorities to interrupt vectors. Interrupts of lower priority are blocked while an interrupt of a higher priority is delivered. Therefore, the hypervisor cannot guarantee that a sent IPI will be handled within a predefined time. Suspending the operating system for long periods can cause the operating system's watchdog timer to trigger a stop error (BSoD).

We present a different method to solve the inter-processor synchronization problem that is based on a documented functionality of the operating system itself. The `KeIpiGenericCall` function (Microsoft Corporation, 2018) receives a callback function as a parameter and executes it on all the active processors simultaneously. We propose to use the `KeIpiGenericCall` function to configure the access rights simultaneously on all the processors.

Because it is impossible to call an operating system function from within the context of the hypervisor, the hypervisor calls the `KeIpiGenericCall` function from the context of the (guest) operating system. In order to achieve this, the hypervisor performs several preparations and then resumes the execution of the operating system.

Algorithm 2 presents three functions that together perform simultaneous access rights configuration on all the active processors. The first function, HANDLECPUID, is part of the hypervisor. This function is called whenever the operating system invokes a special `CPUID` instruction. Two other functions, GUESTENTRY and CALLBACK, are mapped by the hypervisor to a non-occupied region of the operating system's memory.

Algorithm 1 begins with a special `CPUID` instruction, called `FREEZE`. This instruction is handled by lines 2–5 in Algorithm 2: the hypervisor maps GUESTENTRY and CALLBACK, saves the current registers' values and sets the instruction pointer to the address of GUESTENTRY. The GUESTENTRY function calls the operating system's KEIPIGENERICCALL, which will execute CALLBACK on all the active processors. The CALLBACK function performs another special `CPUID` instruction, called `CONFIGURE`, which causes the hypervisor to configure the access rights of all (but the delay-sensitive) memory pages on all the processors. This is handled by lines 6–7 of the algorithm, where we omitted the configuration procedure itself. After the termination of the CALLBACK function, the control returns to the GUESTENTRY function, which executes a special `CPUID` instruction, named `RESUME_OS`. In response, the hypervisor restores the registers' values, which were previously saved in line 4. The operation continues from the instruction following `FREEZE`, which triggered this sequence of events.

Algorithm 2: Simultaneous access rights configuration on all the active processors.

```
1:  function HANDLECPUID(reason)
2:      if reason=FREEZE then
3:          Map GUESTENTRY and CALLBACK
4:          Save registers
5:          RIP ← GUESTENTRY
6:      else if reason=CONFIGURE then
7:          ...
8:      else if reason=RESUME_OS then
9:          Restore registers
10:     else if reason=DUMP then
11:         ...
12:     ...
13: function GUESTENTRY
14:     KEIPIGENERICCALL(CALLBACK)
15:     CPUID(RESUME_OS)
16: function CALLBACK
17:     CPUID(CONFIGURE)
```

5 EVALUATION

This section evaluates the performance of the HV and its memory usage. First, we demonstrate the overall performance impact of the HV. Next, we analyze the memory usage of the HV. Finally, we evaluate the performance of the memory acquisition process.

All the experiments were performed in the following environment:

- CPU: Intel Core i5-6500 CPU 3.20GHz (4 physical cores)
- RAM: 16.00 GB
- OS: Windows 10 Pro x86-64 Version 1803 (OS Build 17134.407)

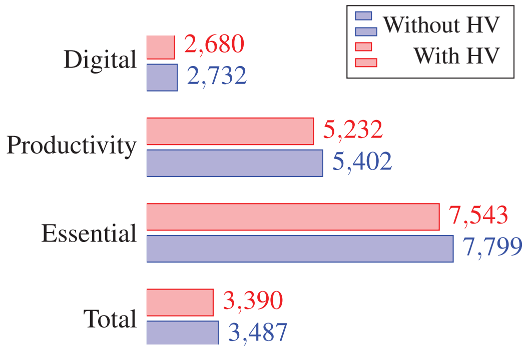


Figure 2: Scores (larger is better) reported by PCMark in four categories: Digital Content Creation, Productivity, Essential, and Total.

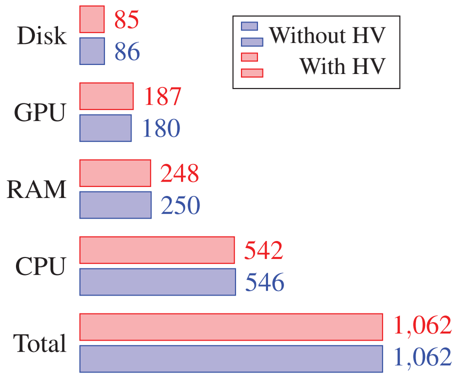


Figure 3: Scores (larger is better) reported by Novabench in five categories: Disk, GPU, RAM, CPU, and Total.

- C/C++ Compiler: Microsoft C/C++ Optimizing Compiler Version 19.00.23026 for x86

5.1 Hypervisor Performance Impact

We start by demonstrating the performance impact of the hypervisor on the operating system. We picked two benchmarking tools for Windows:

1. PCMark 10 – Basic Edition. Version Info: PCMark 10 GUI – 1.0.1457 64 , SystemInfo – 5.4.642, PCMark 10 System 1.0.1457,
2. Novabench. Version Info: 4.0.3 – November 2017.

Each tool performs several tests and displays a score for each test. We invoked each tool twice: with and without the hypervisor. The results of PCMark, and Novabench are depicted in Figures 2–3, respectively. We can see that the performance penalty of the hypervisor is approximately 5% on average.

Table 3: Memory Regions' Sizes.

Index	Name	Size (MB)
0	MiVaUnused	6
1	MiVaSessionSpace	100
2	MiVaProcessSpace	0
3	MiVaBootLoaded	0
4	MiVaPfnDatabase	0
5	MiVaNonPagedPool	6
6	MiVaPagedPool	0
7	MiVaSpecialPoolPaged	5
8	MiVaSystemCache	52
9	MiVaSystemPtes	0
10	MiVaHal	0
11	MiVaSessionGlobalSpace	0
12	MiVaDriverImages	8

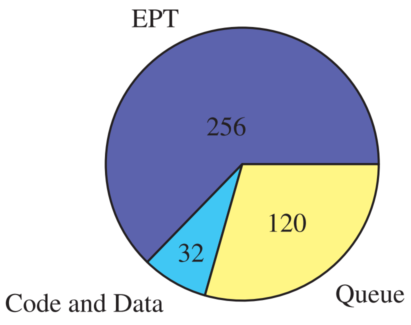


Figure 4: Hypervisor's Memory Usage [MB].

5.2 Memory Usage

The memory used by the hypervisor can be divided into three main parts:

1. the code and the data structures of the hypervisor,
2. the EPT tables used to configure the access rights to the memory pages, and
3. the queue used to accumulate the modified pages.

Figure 4 presents the memory usage of the hypervisor including its division.

The size of the queue is mainly dictated by the number of delay-sensitive pages. Table 3 presents the typical size of each memory region. Pages belonging to the following regions are copied by the hypervisor:

1. MiVaProcessSpace
2. MiVaPagedPool
3. MiVaSpecialPoolPaged
4. MiVaSystemCache
5. MiVaSystemPtes
6. MiVaSessionGlobalSpace

Their total size is $\approx$ 60MB. The size of the queue should be slightly larger than the total size of the

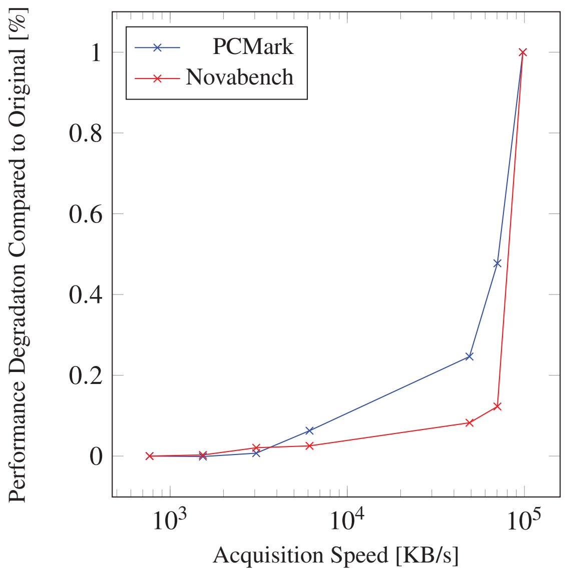


Figure 5: Performance degradation due to memory acquisition.

delay-sensitive pages because regular pages can be modified by the operating system before the content of the queue is exported. Our empirical study shows that it is sufficient to enlarge the queue by 60MB.

5.3 Memory Acquisition Performance

In this section we study the correlation between the speed of memory acquisition and the overall system performance. Figure 5 shows the results. The horizontal axis represents the memory acquisition speed. The maximal speed we could achieve was 97920KB/s. At this speed, the system became unresponsive and the benchmarking tools failed. The vertical axis represents the performance degradation (in percent) measured by PCMark and Novabench. More precisely, denote by $t_i(x)$ the *Total* result of benchmark $i = 1, 2$ (for PCMark and Novabench, respectively) with acquisition speed of x; then, the performance degradation $d_i(x)$ is given by $d_i(x) = 1 - \frac{t_i(x)}{t_i(0)}$.

6 CONCLUSIONS

The method presented in this work should be seen as an incremental improvement over previously described methods, which have similar purpose and design. We describe two improvements over the currently available methods:

1. Our hypervisor supports multiple processors by utilizing an operating system's function for processor synchronization.

2. Our hypervisor supports modern operating systems, e.g., Windows 10, by locating and copying the delay-sensitive pages.

Section 5 presents the memory usage of the hypervisor. We believe that this memory usage can be improved by reducing the number of pages that the hypervisor considers to be delay-sensitive.

REFERENCES

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM.

Ben-Yehuda, M., Xenidis, J., Ostrowski, M., Rister, K., Bruemmer, A., and Van Doorn, L. (2007). The price of safety: Evaluating iommu performance. In *The Ottawa Linux Symposium*, pages 9–20.

Brendmo, H. K. (2017). Live forensics on the windows 10 secure kernel. Master's thesis, NTNU.

Carrier, B. D. and Grand, J. (2004). A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60.

Cohen, M. (2014). Rekall memory forensics framework. *DFIR Prague*.

Durve, R. and Bouridane, A. (2017). Windows 10 security hardening using device guard whitelisting and applocker blacklisting. In *Emerging Security Technologies (EST), 2017 Seventh International Conference on*, pages 56–61. IEEE.

Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. (2016). Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press.

Field, S. (2006). An introduction to kernel patch protection. http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx.

Intel Corporation (2018). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

Kiperberg, M., Resh, A., and Zaidenberg, N. J. (2015). Remote attestation of software and execution-environment in modern machines. In *Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on*, pages 335–341. IEEE.

Kiperberg, M. and Zaidenberg, N. (2013). Efficient remote authentication. In *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*, page 144. Academic Conferences Limited.

Martignoni, L., Fattori, A., Paleari, R., and Cavallaro, L. (2010). Live and trustworthy forensic analysis of commodity production systems. In *International Workshop on Recent Advances in Intrusion Detection*, pages 297–316. Springer.

Microsoft Corporation (2009). Device\PhysicalMemory Object. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc787565(v=ws.10). [Online; accessed 02-Nov-2018].

Microsoft Corporation (2018). KeIpiGenericCall function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-keipigenericcall.

Oracle (2018). VirtualBox. https://www.virtualbox.org/.

Qi, Z., Xiang, C., Ma, R., Li, J., Guan, H., and Wei, D. S. (2017). Forenvisor: A tool for acquiring and preserving reliable data in cloud live forensics. *IEEE Transactions on Cloud Computing*, 5(3):443–456.

Reina, A., Fattori, A., Pagani, F., Cavallaro, L., and Bruschi, D. (2012). When hardware meets software: a bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th annual computer security applications conference*, pages 79–88. ACM.

Resh, A., Kiperberg, M., Leon, R., and Zaidenberg, N. J. (2017). Preventing execution of unauthorized native-code software. *International Journal of Digital Content Technology and its Applications*, 11.

Resh, A. and Zaidenberg, N. (2013). Can keys be hidden inside the cpu on modern windows host. In *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*, page 231. Academic Conferences Limited.

Russinovich, M. E., Solomon, D. A., and Ionescu, A. (2012). *Windows internals*. Pearson Education.

Seshadri, A., Luk, M., Qu, N., and Perrig, A. (2007). Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 335–350. ACM.

Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., and Kato, K. (2009). Bitvisor: A thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 121–130, New York, NY, USA. ACM.

Unified EFI, Inc. (2006). Unified Extensible Firmware Interface Specification, Version 2.6.

VMware (2018). VMware Workstation Pro. https://www.vmware.com/il/products/workstation-pro.html.

Zaidenberg, N. J. (2018). Hardware rooted security in industry 4.0 systems. In Dimitrov, K., editor, *Cyber defence in Industry 4.0 and Related Logistic and IT Infrastructures*, chapter 10, pages 135–151. IOS Press.

Zhang, L., Wang, L., Zhang, R., Zhang, S., and Zhou, Y. (2010). Live memory acquisition through firewire. In *International Conference on Forensics in Telecommunications, Information, and Multimedia*, pages 159–167. Springer.

PIII

HYPERVISOR-BASED PROTECTION OF CODE

by

M. Kiperberg, R. Leon, A. Resh, A. Algawi, N.J. Zaidenberg 2019

IEEE Transactions on Information Forensics and Security, Vol. 4, No. 8, pp. 2203-2216

https://doi.org/10.1109/TIFS.2019.2894577

Hypervisor-based Protection of Code

Michael Kiperberg*, Roee Leon†, Amit Resh‡, Asaf Algawi† and Nezer J. Zaidenberg §
*Faculty of Sciences
Holon Institute of Technology
Israel
Email: michaelkip@hit.ac.il
†Department of Mathematical IT
University of Jyväskylä
Finland
Emails: roee.leonn@gmail.com, asaf.algawi@gmail.com
‡School of Computer Engineering
Shenkar College of Engineering, Design and Art
Israel
Email: amitr44@gmail.com
§School of Computer Science
The College of Management, Academic Studies
Israel
Email: nzaidenberg@me.com

***Abstract*—The code of a compiled program is susceptible to reverse-engineering attacks on the algorithms and the business logic that are contained within the code. The main existing countermeasure to reverse-engineering is obfuscation. Generally, obfuscation methods suffer from two main deficiencies: (a) the obfuscated code is less efficient than the original, and (b) with sufficient effort, the original code may be reconstructed. We propose a method that is based on cryptography and virtualization. The most valuable functions are encrypted and remain inaccessible even during their execution, thus preventing their reconstruction. A specially crafted hypervisor is responsible for decryption, execution and protection of the encrypted functions. We claim that the system can provide protection even if the attacker: (a) has access to the operating system kernel, and (b) can intercept communication over the system bus. The evaluation of the system's efficiency suggests that it can compete with and outperform obfuscation-based methods.**

***Index Terms*—Security, code protection, cryptography, virtual machine monitors, trusted platform module,**

I. Introduction

A compiled program is susceptible to two types of attacks: theft and tampering. The main countermeasure against these attacks is obfuscation, which can be defined as a transformation that produces a more *complex* program but which has the same observable behavior [1]. Several taxonomies classify the different obfuscation methods by the abstraction level (source code, machine code), the unit (instruction, function, program), or the target (data, code) of the performed transformation [2]. According to this classification, the abstraction level of our method is machine code, its unit of transformation is function, and it mainly targets code. Specifically our method does not protect variables [3], [4], [1], [5], the stack [6], [7], [8] or any other information that does not reside within a function.

There is a wide range of approaches to code protection. The simplest forms of code protection are:

- instruction reordering [4], [9], in which independent instructions of the original program are permuted
- instruction substitution [4], [10], [11], in which sequences of instructions are replaced by other but equivalent sequences
- garbage insertion [4], in which the transformation inserts new sequences of instructions that do not affect the execution of the program
- dead code insertion [1], in which the transformation inserts new sequences of instructions that are never executed

All these methods are vulnerable to automatic attacks [12], [13], [14], [15].

A more sophisticated method of code protection is encoding, which either encrypts [16] or compresses portions of the original program and decodes these portions back prior to their execution. However, even when an encryption is used, the cryptographic key is embedded in the decryption algorithms [17], [18], [19], and therefore can be extracted [20], [21], [22]. Moreover, since the code eventually must be decrypted, it can be extracted during run-time. Therefore, such methods are usually combined with run-time analysis prevention methods [23], [24].

A particular case of encoding is virtualization, in which the program is translated to a different instruction set, and then executed by a special embedded interpreter [4], [25]. Automatic [26] and semi-automatic [27] attacks have been proposed for this method.

The performance degradation due to obfuscation depends on the sophistication of the obfuscation method. For example, the Obfuscator-LLVM [28] specifies which obfuscation techniques should be applied to an executable. When only instruction substitution is applied, the performance penalty is $\approx 12\%$ on average. However, when additional techniques are added, e.g. bogus control flow, control flow flattening, function annotations, etc., the execution times increase by a factor of 15–35. Stunnix [29] and Tigress [30] produce executables that are slower by a factor of ≈ 9 [31].

Our method can be classified as encoding, since it encrypts a set of functions and then decrypts them prior to their execution. However, there are two advantages to our method over current methods:

1) In our method, the decryption key is not embedded in the decryption algorithm, but rather the key is stored in a widely available hardware device (TPM).
2) In our method, the decrypted code is protected by a hypervisor, during its execution, thus making the method safe even in presence of a run-time analysis tool.

We show that the performance degradation of our method is 5%-25% on average, depending on an application.

A hypervisor is a software module that can monitor and control the execution of an operating system. The monitoring and controlling capabilities are provided by an extension to the original processor's instruction set, called a virtualization extension. Virtualization extensions are available on processors designed by Intel (VT-x) [32], AMD (AMD-V) [33] and ARM [34]. Our method is implemented on Intel processors but can easily be ported to AMD and ARM.

Trusted Platform Module (TPM) [35] is a standard that defines a device with a non-volatile memory and a predefined set of cryptographic functions. The system described in this paper uses the TPM to store the decryption key (by sealing and unsealing it).

Throughout this paper we refer to the entity that wants to protect the program as the *distributor*, and to the potentially malicious entity that uses the program as the *user*.

A. VMX

Many modern processors are equipped with a set of extensions to their basic instruction set architecture that enables them to execute multiple operating system simultaneously. This paper discusses Intel's implementation of these extensions, which they call Virtual Machine Extensions (VMX). The software that governs the execution of the operating systems is called a *hypervisor* and each operating system (with the processes it executes) is called a *guest*. Transitions from the hypervisor to the guest are called VM-entries and transitions from the guest to the hypervisor are called VM-exits. While VM-entries occur voluntarily by the hypervisor, VM-exits are caused by events that occur during the guest's execution. The events may be synchronous, e.g. execution of `INVLPG` instruction, or asynchronous, e.g. page-fault or general-protection exception. The event that causes a VM-exit is recorded for future use by the hypervisor. A special data structure called Virtual Machine Control Structure (VMCS) allows the hypervisor to specify the events that should trigger a VM-exit, as well as many other settings of the guest.

Intel's Extended Page Table (EPT), a technology generally called Secondary Level Address Translation (SLAT), allows the hypervisor to configure a mapping between the physical address space, as it is perceived by a guest, to the real physical address space. Similarly to the virtual page table, EPT allows the hypervisor to specify the access rights for each guest physical page. When a guest attempts to access a page that is either not mapped or has inappropriate access rights, an event called an EPT-violation occurs, triggering a VM-exit.

Input-Output Memory Management Unit (IOMMU) allows the hypervisor to specify the mapping of the physical address space as perceived by the *hardware devices* to the real physical address space. It is a complementary technology to the EPT that allows a construction of a coherent guest physical address space for both the operating system and the devices.

B. System Description

The system described in this paper consists of an UEFI application and an encryption tool. The encryption tool allows the distributor to encrypt a set of selected functions in a given program. The UEFI application initializes a hypervisor that enables the execution of encrypted functions by running the operating system (and all its processes) as a guest. An encrypted program starts executing as usual but whenever it jumps to an encrypted function, a VM-exit occurs. The hypervisor decrypts the function and executes the decrypted function in user-mode (in the context of the hypervisor). When the function returns, the hypervisor performs a VM-entry and the normal program execution continues.

The main benefit of using a hypervisor is its ability to construct an isolated environment, which is inaccessible from the outside of the hypervisor. Like the hypervisor, the operating system can also construct an isolated environment. In principle, the system described in this paper can be realized as a module in an operating system. However, the security of this module will depend on the security of all the other code that executes in kernel mode: the operating system and the device drivers. Studies show [36] that the number of software defects increases with the size of the software. Some of these defects (1%–2% [37]) can be classified as security vulnerabilities. The size of operating system varies between 20 to 50 million lines of code [37]. Moreover, since every hardware vendor can produce a device driver that executes in kernel mode, the size of the code executing in kernel mode is unlimited. In contrast, the size of the hypervisor presented in this paper is 10,000 lines of code, which makes it a much better candidate to provide the described security guarantees.

The UEFI application uses the TPM to unseal the decryption key, which is stored (in its sealed form) in a local file. After activating the hypervisor the (unsealed) key is delivered to the hypervisor through a secure communication channel. The configuration of EPT and IOMMU does not map the pages that contain the key and the decrypted functions, making them inaccessible both from the guest and from a hardware device.

C. Threat Model

We argue that the system described in this paper can withstand the following types of attacks:

- malicious code executing in user-mode or kernel-mode,
- malicious hardware devices connected via a DMA controller equipped with IOMMU,
- sniffing on any bus.

More precisely, we claim that even in the presence of all the above types of attacks, the attacker cannot obtain the decryption key nor the decrypted functions.

We admit that the system is vulnerable to attacks that:

1) broadcast on buses or
2) move the TPM to another (malicious) machine.

Finally, we assume that the firmware, including the code that executes in SMM [38], is trustworthy.

Since the protocol by which the secret key is delivered to the TPM is not covered by this paper, and in order to make the description herein self-contained, we assume that the TPM already contains the secret key needed for code decryption.

We admit that key distribution is not required in obfuscation-based methods. If the complexity of key distribution is unacceptable, obfuscation-based methods are the preferred choice.

We further note that the safety of the hypervisor is guaranteed by its design, which is presented in this paper. We do not impose any limitations on the attacker with this regard. Specifically, an attacker's inability to execute code in the hypervisor is not an assumption but rather a consequence of the hypervisor's design.

II. Related Work

The idea of utilizing a hypervisor's ability for creating an isolated environment for security applications is not new. Hypervisors were used for integrity verifications [39], [40], [41], for creating isolated domains inside an operating system [42], for creating malware [43], and for detecting and analyzing malware [44]. However, to the best of our knowledge, hypervisors were never used for code protection.

There is a wide range of approaches to code protection. Among the different methods, from an operational point of view the system described in this paper is most closely related to *encoding*-based methods. However, the security guarantees of the system are comparable to those given by *obfuscation*-based methods.

Encoding-based methods encrypt portions of a program and the decrypt them prior to their execution. The decryption key is embedded in the decryption algorithms and therefore can be extracted. Moreover, since the code eventually must be decrypted, it can be extracted during run-time. A notable advantage of these methods is their performance. Since the encrypted code is decrypted only once, during the program's loading, the overhead of this protected method is minimal. Commercial examples of this method, PELock [45] and UPX [46], indeed show a negligible overhead but are vulnerable to automatic code extraction [47].

The security guarantees of obfuscation-based methods can be summarized as follows: "increase the reverse-engineering costs in a sufficiently discouraging manner for an adversary" [28, p. 1]. This guarantee is achieved by applying various transformations to the program that should be protected. There are numerous transformations ranging from basic instruction substitutions, which replaces $a = b + c$ by $a = b - (-c)$, to control-flow flattening, which reorganizes the flow between basic blocks of a function. Obviously, applying several transformations together improves the security but degrades the performance of the system. Stunnix [29] is a commercial source code obfuscator for different programming languages, including C++, Perl and JavaScript. Tigress [30] is an open-source C-language obfuscator/virtualizer. Obfuscator-LLVM [28] is an open-source obfuscator that works on LLVM bitcode. It can obfuscate programs written in all the languages supported by LLVM. Obfuscator-LLVM also implements a wide variety of obfuscating transformations. The execution time of programs protected by Obfuscator-LLVM can increase by a factor of 15–35. Stunnix and Tigress produce executables that are slower by a factor of ≈ 9 [31].

In comparison to the methods described above, we argue that our system provides stronger security guarantees than both methods. Our system outperforms Obfuscator-LLVM when a reasonable set of transformations is applied.

III. The Hypervisor

The main component of the described system is a hypervisor, which utilizes the VMX instruction set extension. This section provides a short overview of this component. Section V-A contains a detailed description of the hypervisor's initialization and operation. There are two types of hypervisors: full hypervisors and thin hypervisors. Full hypervisors like Xen[48], VMware Workstation [49], Oracle VirtualBox [50] can execute several operating systems concurrently. The main goal of VMX was to provide software developers with means to construct efficient full hypervisors. Thin hypervisors, in contrast, can execute only a single operating system. Their main purpose is to enrich the functionality of an operating system. The main benefit of a hypervisor over kernel modules (device drivers) is the hypervisor's ability to create an isolated environment, which is important in some cases. For example, SecVisor [39] is a thin hypervisor that validates an operating system's integrity, TrustVisor [51] is a thin hypervisor that provides code and data integrity and secrecy services to user mode applications, and BitVisor [52] is a thin hypervisor that encrypts data that is transmitted to hard drives. In general, since thin hypervisors are much smaller than full hypervisors, they are superior in their performance, security and reliability. The hypervisor described in this paper is a thin hypervisor that is able to:

1) intercept attempts to execute an encrypted function,
2) decrypt the function,
3) execute the function.

The hypervisor was written from scratch to achieve an optimal performance.

Similarly to an operating system, a hypervisor does not execute voluntarily but responds to events, e.g. execution of special instructions, generation of exceptions, access to memory locations, etc. The hypervisor can configure interception of (almost) each event. Interception of an event (a VM-exit) is similar to handling of an interrupt, i.e. a predefined function is executed by the processor. Another similarity with an operating system is the hypervisor's ability to configure the access rights to each memory page through a data structure, named EPT, that resembles the virtual page table. The configuration that is specified in EPT is activated when the processor leaves the code of the hypervisor and is deactivated when the processor switches to the hypervisor to handle an event. Therefore, the hypervisor can configure a page to be accessible only by the hypervisor by marking the page as inaccessible in the EPT.

Our hypervisor operates as follows. During its initialization, the hypervisor allocates a memory region and configures the EPT to make this region inaccessible. Then, the hypervisor configures interception of attempts to execute encrypted functions (actually, as explained in section V-B this is done by intercepting a special exception). Finally, the hypervisor boots the operating system. The hypervisor remains passive until the operating system loads a protected program and an encrypted function is executed. The encrypted function generates an exception which is intercepted by the hypervisor. The hypervisor decrypts the encrypted function to the pre-allocated and protected memory region. The decrypted function executes inside the hypervisor and upon completion returns to the original program (outside the hypervisor).

IV. Preparations

In order to protect her code, the distributor has to encrypt the sensitive code and install the necessary files on a target machine. These processes are described in the following paragraphs.

A. Encryption

Encryption is performed in a granularity of a function. It receives as input a text file that specifies the executable files and the functions to be encrypted as well as the encryption key to be used.

The encryption tool produces an output file that contains the encrypted versions of all the functions that were selected for encryption. We call this output file a *database*. In addition to the generation of the *database* file, the encryption tool "erases" the instructions of the selected functions. The erasing of code is performed by replacing the original instructions by a special instruction. We call the resulting file a *protected executable*.

The special instruction we have chosen is the `HLT` instruction. When executed in kernel mode, the `HLT` instruction causes the processor to halt. In user mode however, this instruction generates a general protection exception, which can be intercepted by the hypervisor. Any instruction that generates an exception can qualify as a special instruction, in this sense. For example, the `INT3` instruction generates a breakpoint exception and therefore can be used to replace the functions' original instructions.

The encryption tool is implemented as a command line program. It can be invoked from a standard makefile rule or as a post-build action, thus allowing the distributor to automate the encryption process.

The system described in this paper supports simultaneous execution of only a single protected executable, or to be precise, a single protected process. We plan to add support for multiple processes in the future.

B. Target Installation

In UEFI enabled systems, after a successful initialization, the firmware loads a sequence of executable images, called UEFI applications. The sequence is stored in a firmware-defined non-volatile storage. Each element in the sequence points to a location that contains an UEFI application. After loading an UEFI application to the memory, the firmware calls the application's *main* function. If the *main* function returns, the firmware loads the next application and so on. Typically, the operating system's boot loader is implemented as an UEFI application, whose *main* function does not return. The firmware settings screen allows the boot sequence to be configured.

The system described in this paper is implemented as a UEFI application. In order to install the system, the distributor should perform the following steps:

1) place the UEFI application at a location accessible by the firmware: local disk, USB device, TFTP server or possibly others,
2) modify the boot sequence so that the application is pointed by the first element of the boot sequence,
3) store the code decryption key by performing the boot process.

During its first execution, the application asks the user to enter the code decryption key. Then, the application encrypts the key using a TPM, a process called *sealing*, and stores the resulting encrypted key in a local file.

During subsequent executions, the application reads the file and decrypts its contents using the TPM, thus obtaining the code decryption key. As will be explained in section VI, it is impossible to obtain the code decryption key from another UEFI application.

The distributor should store the *database* file, which was produced during the encryption phase, on a local drive that is accessible by the firmware. We recommend the use of the ESP (EFI System Partition). The ESP can be mounted and become a regular folder, which simplifies updates of the configuration file after the initial provisioning.

We note that it is possible to deliver the key to the TPM from a remote entity. There are several protocols and technologies [53], [54], [55] that allow one entity to prove its authenticity and obtain a key from a remote entity. This topic however is beyond the scope of this paper.

V. Operation

The UEFI application, during its execution, obtains the code decryption key using the TPM, loads the *database* file, initializes a hypervisor, and returns to firmware. The firmware typically proceeds by loading the operating system boot loader. The hypervisor remains in the main memory and continues its operation even after the application terminates. The hypervisor is responsible for detecting attempts to execute encrypted

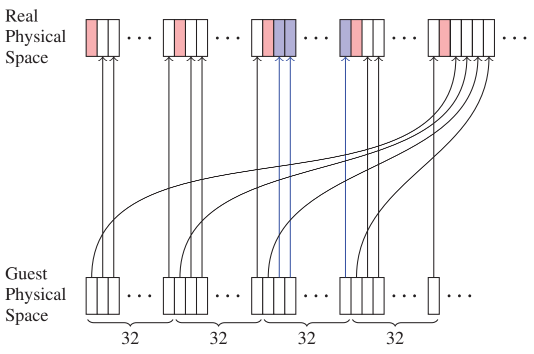


Fig. 1. EPT configuration that maps guest physical pages (bottom) to real physical pages (top). Real physical pages whose index is a multiple of 32 (red rectangles) are safe pages, which are used exclusively by the hypervisor. Guest's physical pages whose index is a multiple of 32 are mapped to the last pages in the real physical address space. The code and the data of the hypervisor (blue rectangles) are mapped as read-only (blue lines) to the guest physical address space.

functions (whose instruction were replaced by a special instruction). When such an attempt is detected, the hypervisor decrypts the function, and executes it on behalf of the original program. During this execution, the hypervisor protects the decrypted version of the function from being exposed. The rest of this section provides a detailed explanation about the initialization and the operation of the system.

A. Initialization

The UEFI application starts by allocating a persistent memory block (a memory block that can be used after the application terminates), and loading the *database* file into this memory block. Fortunately, file reading is one of the services provided by the UEFI firmware.

Next, the UEFI application loads the encrypted key from a file and decrypts it using the TPM. The communication with the TPM is carried over a secure channel, thus eliminating man-in-the-middle attacks. When the decryption is completed, the application forces the TPM to transit to a state in which it is no longer possible to decrypt the key file.

Finally, the UEFI application allocates a persistent memory block, and initializes a hypervisor. During the hypervisor's initialization, the EPT and IOMMU are set up. The EPT defines a mapping between physical addresses as perceived by the operating system, and the real physical addresses. In this sense, EPT is similar to the page tables that map virtual addresses to physical addresses. For this reason, EPT is called a secondary level address translation (SLAT). IOMMU defines a mapping between physical addresses as perceived by hardware devices and the real physical addresses. Both EPT and IOMMU define not only the mapping of the perceived addresses but also their access rights.

Fig. 1 depicts the mapping that the hypervisor establishes during its initialization. The mapping organization chases two goals.

1) The first goal is protection of the hypervisor's code and data from malicious modification. This goal is achieved by setting the access rights of the hypervisor's code and data to be read-only.
2) The second goal is protection of decrypted code from cache eviction attacks. Section VI contains a detailed discussion of this attack and its prevention. Here, we just note that this goal is achieved by a technique called page-coloring. In essence, this technique allows the reservation of some portion of the processor's cache to be used exclusively by the hypervisor. This reservation is performed by excluding all pages whose index is a multiple of 32 from the mapping. Thus, the portion of the cache that backs those pages cannot be affected by malicious code executing inside the guest.

Before establishing the new mapping, the hypervisor copies the contents of pages whose index is a multiple of 32 to the pages that correspond to them in the mapping. In order to understand the necessity of this step, consider the following scenario. The firmware stores some value in page 32 before the hypervisor's initialization, and loads this value from page 32 after the initialization. Let us assume that page 32 is mapped to page 10032. The first access will store some value to physical page 32. The second access however will load the value from page 10032. Therefore, the contents of page 32 must be copied to page 10032. The hypervisor uses the pages whose indexes are a multiple of 32 to store sensitive information, like the code of decrypted functions and the decryption key. That is why we call these pages *safe pages*.

B. Transitions

A *protected executable* can run as usual without any interference while only functions that were not selected for encryption are called. The hypervisor silently waits for an encrypted function to be called. Recall that the encryption tool replaces the original instructions of a function that was selected for encryption by a special instruction that generates an exception. The hypervisor is configured to intercept that specific kind of exception, namely the general protection exceptions. When such an exception is generated, a VM-exit occurs. The processor saves the guest's state to VMCS, loads the hypervisor's state from VMCS, and begins execution of the hypervisor's defined VM-exit handler. The handler checks whether the general protection exception was caused by execution of an encrypted function. If not, the hypervisor injects the exception to the guest, thus delegating the exception handling to the operating system. If the hypervisor detects an attempt to execute an encrypted function, it locates, in the *database*, the encrypted version of this function, which was loaded during the hypervisor's initialization. Then, the hypervisor decrypts the function to one or more *safe pages*. Finally, the hypervisor makes some preparations and jumps to the decrypted function.

In order to understand the nature of the preparations that were mentioned in the previous paragraph, we shall discuss the virtual address space of the hypervisor. Fig. 2 illustrates the virtual address space layouts of the hypervisor and the guest. In the x86-64 instruction set, code and memory accesses are instruction-relative. This means that the same sequence

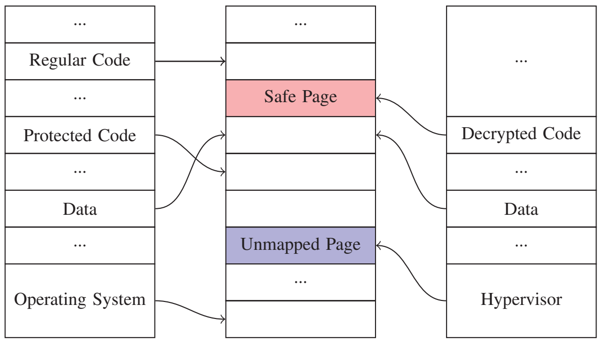


Fig. 2. Virtual address space layouts of the hypervisor and the guest during protected function execution. The code and the data structures of the hypervisor are not mapped in the guest. The protected code is decrypted to a safe page. The virtual address of the protected function in the hypervisor corresponds to its virtual address in the guest. The mapping of the pages that store data are identical in the hypervisor and the guest. The code of the operating system is not mapped in the hypervisor.

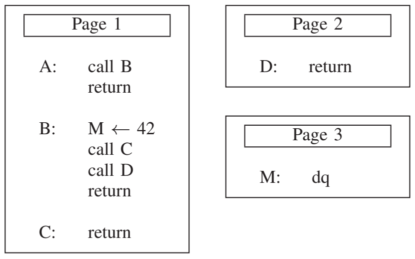


Fig. 3. Sample program. The program consists of four functions (A, B, C, D) and of one variable (M). The functions A, B and C reside in the first page. The function D resides in the second page. The variable M resides in the third page. In this program only the function B is encrypted.

of instructions will give different results if executed from different virtual addresses. Therefore, it is highly important to execute the decrypted functions from their natural virtual addresses. Usually operating systems divide the virtual address space into two large regions. In 64-bit Windows, the upper half of the 64-bit space contains the code and data of the kernel, while the lower half contains the code and data of a process. The hypervisor mimics this behavior by holding its code and data in the upper half of its virtual address space; the lower half is reserved for decrypted functions and their data. Whenever the hypervisor decrypts a function to a *safe page*, it maps this safe page such that the virtual address of the decrypted function equals to the virtual address of the protected function. Finally, the hypervisor transitions to user-mode (under context of the hypervisor) and jumps to the decrypted function. (These two operations are performed by a single `IRET` instruction.)

The execution of the decrypted function continues until it generates an exception. The hypervisor can handle some exceptions; others are injected to the operating system. During its execution, a decrypted function, can attempt to read from, or write to, a page that is not mapped in the hypervisor's virtual address space. In such a case, the hypervisor will copy the corresponding mapping from the operating system's virtual address space. When the decrypted function completes and the hypervisor returns to the guest, the mappings that were constructed are retained for future invocations of that function. However, the operating system is free to reorganize its mappings (e.g. due to paging), thus making the hypervisor's mapping invalid. The handling of different cases is described in Algorithm 1.

The algorithm sketches the implementation of the hypervisor's VM-exit handler. Each VM-exit is caused by some condition that occurred in the guest (or during the guest's execution). The main condition that the hypervisor intercepts is a general protection exception. General protection exceptions can occur either due to execution of a `HLT` instruction or for some other reason. The hypervisor should provide a special handling only for the first case (lines 3–16); in the second case, the hypervisor should inject the exception to the operating system (lines 1–2).

Algorithm 1 Hypervisor's VM-exit handler

```
1:  if #GP and RIP is not in protected function then
2:      Inject and return to guest
3:  else if #GP and RIP is in protected function then
4:      while TRUE do
5:          Enter user-mode at RIP and await interrupts
6:          if #GP or #PF[INSTR] then
7:              if RIP not in protected function then
8:                  Return to guest
9:              if RIP is not mapped then
10:                 Allocate a safe page & fill it with HLTs
11:                 Map the page to RIP
12:             Decrypt function at RIP
13:         else if #PF[DATA] and mapped in guest then
14:             Copy mapping
15:         else
16:             Inject and return to guest
17: else if INVLPG then
18:     Clear virtual table
19:     Return to guest
```

The hypervisor reacts to a special instruction-induced general protection exception by entering a loop (line 4–16). At the beginning of each iteration (line 5), the hypervisor transitions to user-mode (without returning to guest) and sets the instruction pointer to the same address that generated the general protection exception. The execution continues until an exception occurs in user-mode. During execution of regular functions or handling of interrupts and exceptions, the operating system may modify the mappings of virtual pages. The hypervisor may have copies of some of these mappings, which were modified by the operating system. Thus, it is essential for the hypervisor to intercept all such modifications. Fortunately, according to Intel's specification [32], since the processor stores portions of mapping information in its caches (TLBs), the operating system is required to inform the processor of all modifications through a special instruction, `INVLPG`. The hypervisor intercepts this instruction, and responds to it by erasing all the entries that were copied from the operating system (lines 17–19).

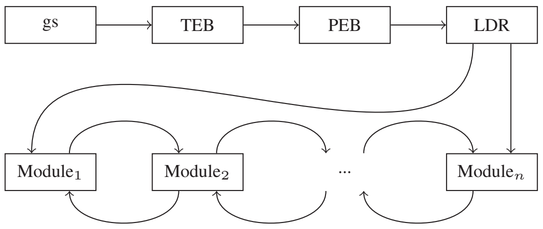


Fig. 4. The path to the executable images of the current process. The special register "gs" points to a data structure that describes the current thread (TEB). The TEB data structure points to a data structure that describes the current process (PEB). The PEB data structure points to a data structure that represents a linked list of all the modules (programs and libraries) loaded by the current process.

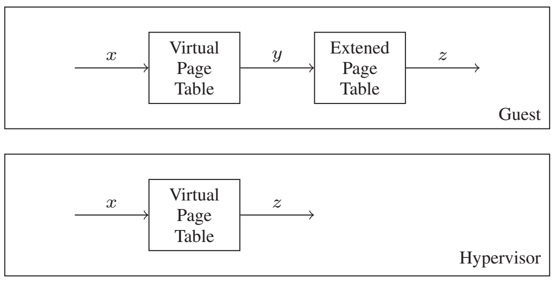


Fig. 5. Address translations in the guest and the hypervisor. When the guest accesses a virtual address x, it is translated to a guest physical address y via the guest's virtual page table, then y is translated to a real physical address z via EPT. When the hypervisor accesses a virtual address x, it is translated to a real physical address via the hypervisor's virtual page table.

C. *OS Dependence*

During some VM-exits the hypervisor needs to check whether the instruction pointer resides in a protected function (line 7 in Algorithm 1). The check is performed in three steps:

1) Firstly, the hypervisor finds the so called *base address* — the address at which the protected program was loaded.
2) Then, the hypervisor calculates the offset of the instruction pointer from the base address,
3) Finally, the hypervisor searches for a function with the calculated offset in the *database*.

In order to find the base address, the hypervisor uses the operating system's data structures. We describe the data structures that are used in 64-bit versions of Windows (see Fig. 4). In Windows, the segment register `gs` points to a data structure called the Thread Environment Block (TEB), which reflects information about the currently executing thread. The TEB contains a field that points to a data structure called the Process Environment Block (PEB), which reflects information about the currently executing process. The PEB contains a field that points to a data structure (LDR) that represents a linked list of all executable images (programs and libraries) loaded by the process. Each entry in this linked list contains the name of the image, its base and its size, thus allowing us to find an image that contains a specific address. Some of these data structures may be paged-out by the operating system from the main memory to the disk. Therefore, the hypervisor may need to force the operating system to load these data structures to the main memory. This is done, by injecting artificial page-fault exceptions to the guest.

VI. Security

We argue that the described system can withstand the following types of attacks:

- malicious code executing in user-mode or kernel-mode,
- malicious hardware devices connected via a DMA controller equipped with IOMMU,
- processor-memory bus sniffing.

More precisely, we claim that even in presence of all the above types of attacks, the attacker cannot obtain the decryption key nor the decrypted functions.

We admit that the system is vulnerable to attacks that:

1) broadcast on buses or
2) move the TPM to another (malicious) machine.

Finally, we assume that the firmware, including the code that executes in SMM, which is more privileged than a hypervisor, is trustworthy. If an attacker is able to broadcast on the CPU–memory bus, he can modify the memory directly; neither EPT nor IOMMU can stop him. Finally, consider an attacker that can move the TPM to another machine where he can install malicious firmware. The firmware can execute the normal sequence of `Extend`s, thus the "correct" values in the PCRs. At this stage the firmware can `Unseal` the decryption key.

A. *Memory Protection*

Secondary level address translation (SLAT), or extended page table (EPT) according to Intel's terminology, is a mechanism that allows hypervisors to control the mapping of physical page addresses as they are perceived by the operating system to the real physical addresses. In this sense, SLAT is similar to virtual page tables, a mechanism that allows the operating system to control the mapping of addresses as they are perceived by the process to the real physical addresses. When a process in a virtualized environment attempts to load a variable at (a virtual) address x, this address is first translated by the virtual page tables to a guest physical address y, then y is translated by EPT to a (real) physical address z (see Fig. 5). The virtual page table is configured by the operating system, while the EPT is configured by the hypervisor.

Input-output memory management unit (IOMMU) is an additional memory translation mechanism. In contrast to virtual page tables and EPT, IOMMU translates addresses that are accessed not by the processor but by hardware devices. Interestingly, the configuration tables of EPT and IOMMU are almost identical.

Protection from malicious access (reading and writing) to the hypervisor's code and data, as well as to the code of decrypted functions is carried out via a special and identical configuration of the EPT and IOMMU mapping. According to this configuration, all the sensitive memory regions are not mapped and are therefore inaccessible from the guest or from a hardware device. The EPT and IOMMU mapping provide the first two security guarantees.

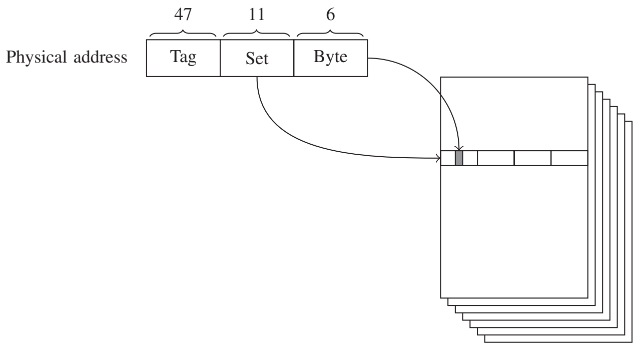


Fig. 6. An example of a last level cache organization of associativity 4 with 6 slices and 2048 sets in each slice. The 6 least significant bits of the physical address select the byte in the cache line. The next 11 bytes select the set. The slice is determined by applying a hash function to the set and the tag fields of the physical address.

We note that since all the hypervisor's memory is allocated via a call to the UEFI memory allocation function, a regular non-malicious operating system will obey this allocation and will not attempt to access this memory region. However only the EPT prevents the operating system from doing so.

The EPT allows the hypervisor to hide the decrypted functions from the operating system and applications including dynamic analysis tools, like debuggers. Likewise, the system, utilizing regular countermeasures (NX bit) [56], is immune to most of the code injection attacks. However the system is prone to more sophisticated attacks, like return-oriented programming, that use the code of the decrypted functions themselves to achieve a malicious behavior. In our case the attacker cannot study the original code, since it is encrypted. Therefore, we believe that it is much harder to carry out a successful attack of this type.

B. Cache Evictions

The third security guarantee, protection from bus sniffing, is much more challenging to achieve. We want to ensure that the sensitive data, the decryption key and the decrypted functions, are never transmitted over the bus. In other words, the sensitive data should reside in the processor's caches at all times.

The cache of Intel processors has three levels (see Fig. 6): the first is the fastest but the smallest, and the last is the slowest and the largest. The last level cache (LLC) is divided into several *slices* of equal size. Each slice has 2048 *sets* and each set has multiple 64-bytes lines. (The number of lines in a set is called *associativity*.) The processor evicts data from the cache to the main memory either as a response to a special instruction (e.g. `WBINVD`), which can be intercepted by the hypervisor, or in order to store some new data. When an instruction accesses the physical address x, the processor determines the cache set in which the data at address x should be stored: the slice is determined by applying an unknown hash function on bits 6–63 of x, the set number is determined by bits 6–16 of x (bits 0–5 determine the bytes number inside the cache line). After the set is determined, the processor selects one of the lines in the set for eviction. We note that access to address x can cause eviction of data from address y only if x and y are mapped to same set. Each page is mapped to $4096/64 = 64$ consecutive sets and only the $2048/64 = 32$th page following it will potentially (since there is more than one slice) be mapped to the same sets. This observation allows the hypervisor to protect its sensitive data (namely the decryption key and the decrypted functions) from cache eviction attacks, by reserving the pages $0, 32, 64, 96, \ldots$ for its own use. The idea of memory allocation that takes into account the cache layout is not new, but it was implemented previously mainly for performance considerations [57], [58].

The size of the cache limits the total size of the decrypted functions. Current processors are equipped with at least 8192KB L3 cache [59], which translates to a limit of $8192/32 = 256$KB. This limitation can decrease the performance significantly, since when the (relevant portion of the) cache becomes full, the hypervisor is forced to erase previously decrypted functions, and then eventually to decrypt them again. Note, however, that the limitation is imposed not on the total size of all the encrypted functions, but on the total size of the functions that participate in a single call sequence.

C. Decryption Key

Trusted Platform Module (TPM) is a standard that defines a hardware device with a non-volatile memory and predefined set of cryptographic functions. The device itself can be implemented as a standalone device mounted on the motherboard, or it can be embedded in the CPU packaging [60, p. 139]. Each TPM is equipped with a public/private key-pair that can be used to establish a secure communication channel between the CPU and the TPM. The non-volatile memory is generally used to store cryptographic keys. The processor of a TPM can decrypt data using a key stored in its memory without transmitting this key on the bus.

One of the main abilities of the TPM is environment integrity verification. The TPM contains a set of Platform Configuration Registers (PCRs) that contain (trustworthy) information about the current state of the environment. These registers can be read but cannot be assigned. The only way to modify the value of these registers is by calling a special function, `Extend`(D) that computes a hash of the given value D and the current value of the PCR, and sets the result as the new value of PCR. The UEFI firmware is responsible for initializing the PCRs and for extending them with the code of UEFI application before jumping to these applications. An application that wants to check its own integrity can compare the relevant PCR with a known value.

Another important ability of the TPM is symmetric cryptography. The TPM provides two functions: `SEAL` and `UNSEAL`. The first function encrypts a given plaintext and binds it to the current values of the PCRs. The second function decrypts the given ciphertext but only if the PCR values are the same as when the `SEAL` function was called.

Algorithm 2 UEFI Application Initialization Sequence

```
1: if FileExists("key.bin") then
2:     $EncryptedKey \leftarrow$ FileRead("key.bin")
3:     $Key \leftarrow$ Unseal($EncryptedKey$)
4: else
5:     $Key \leftarrow$ Input()
6:     $EncryptedKey \leftarrow$ Seal($Key$)
7:     FileWrite("key.bin", $EncryptedKey$)
8: Extend(0)
```

Algorithm 2 presents the initialization sequence of our UEFI application. The application first checks whether a file named ”key.bin” already exists (line 1). If so, its contents are read and unsealed producing a decryption key (line 2–3), which is then stored in a safe page. If, on the other hand, the ”key.bin” file is missing, the application asks the user to type the key (line 5), which is then sealed and stored in a file (lines 6–7). In any case, the PCRs are extended with a (meaningless) value (line 9), thus preventing other UEFI applications and the operating system to unseal the contents of the ”key.bin” file.

The `UNSEAL` function is executed by the TPM and its result is transmitted to the CPU over the bus. In order to protect the decryption key in the presence of a bus sniffer, our UEFI application establishes a secure communication channel (OIAP session). During the initialization of the channel, the application encrypts the messages using the public part of the key that is embedded in the TPM.

VII. Performance

The performance of the described system depends greatly on the set of encrypted functions. The amount of the intellectual property contained within a program may vary, and so does the set of encrypted functions. We note that the performance is affected not only by the amount of encrypted functions, but also by the interconnections between these functions. This fact complicates the performance evaluation, since the functions to be encrypted are not known. Our evaluation is, therefore, randomized, in part.

The evaluation presented below answers the following questions:

1) How the mere existence of the hypervisor degrades the performance?
2) How our system compares to obfuscation with respect to performance?
3) What is the expected performance degradation when X% of a program is encrypted?
4) To what extent an initially poor performance can be improved?

The first question was answered by executing multiple unencrypted benchmarking tools on a system with an active hypervisor. For the second question, we protected the same program using our system and using Obfuscator-LLVM and measured the performance overhead. In order to answer the third question we performed a randomized experiment, during which function sets of different sizes were randomly selected for encryption. Finally, for the fourth question, we show that

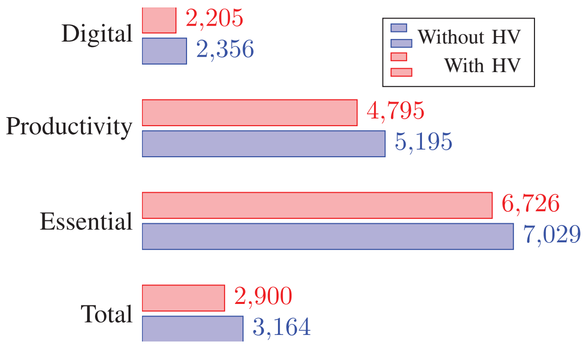


Fig. 7. The scores (larger is better) reported by PCMark in 4 categories: Digital Content Creation, Productivity, Essential and Total.

the hypervisor’s built-in profiler can be used to improve an initially poor performance by two orders of magnitude.

All the experiments were performed in the following environment:

- CPU: Intel Core i5-4570 CPU @ 3.20GHz (4 physical cores)
- RAM: 8.00 GB
- OS: Windows 10 Pro x86-64 Version 1709 (OS Build 16299.248)
- C/C++ Compiler: Microsoft C/C++ Optimizing Compiler Version 19.00.23026 for x86

A. *Hypervisor Performance Impact*

We start by demonstrating the performance impact of the hypervisor on the operating system. We picked three benchmarking tools for Windows:

1) PCMark 10 – Basic Edition. Version Info: PCMark 10 GUI – 1.0.1457 64 , SystemInfo – 5.4.642, PCMark 10 System 1.0.1457,
2) PassMark Performance Test. Version Info: 9.0 (Build 1023) (64-Bit),
3) Novabench. Version Info: 4.0.3 November 2017.

Each tool performs several tests and displays a score for each test. We invoked each tool twice: with and without the hypervisor. The results of PCMark, PassMark and Novabench are depicted in Fig. 7, 8 and 9, respectively. We can see that the performance penalty of the hypervisor is approximately 5% on average.

B. *Comparison*

In this test, we analyze the Obfuscator-LLVM [28] performance impact compared to our method. We cloned the latest Obfuscator-LLVM directly from the official Git repository and built a 32-bit version. For the comparison, we protected 7-Zip using Obfuscator-LLVM and using our system. Specifically, we tested the ”extracting files from an archive (the `e` command line option)” and used a 7z compressed tarball of the latest Linux kernel to date (4.15.6).

We performed two tests which differed in the set of functions that was selected for protection. In the first test the set included the functions `DecodeToDic`,

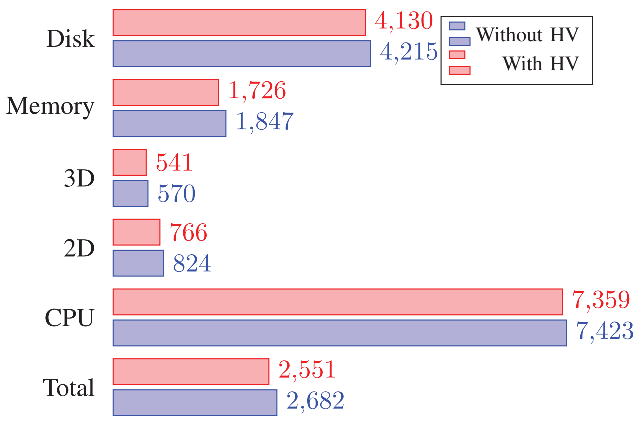


Fig. 8. The scores (larger is better) reported by PassMark in 6 categories: Disk, Memory, 3D, 2D, CPU and Total.

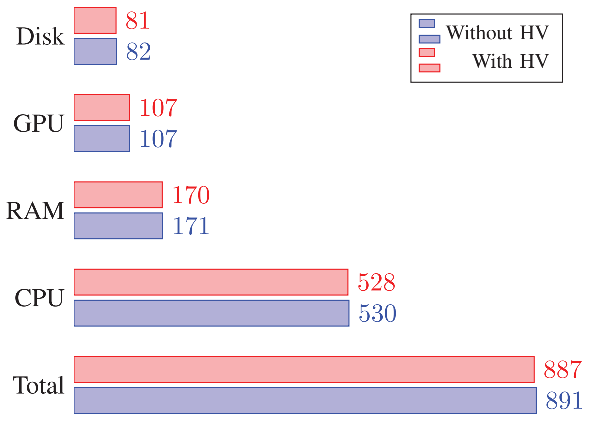


Fig. 9. The scores (larger is better) reported by Novabench in 5 categories: Disk, GPU, RAM, CPU and Total.

`DecodeReal2,WriteRem` that constitute $\approx 1\%$ of the total execution time. In the second test we added `DecodeReal` to the set of functions that now constitute $\approx 84\%$ of the total execution time. In both tests the functions were encrypted using our system and obfuscated using Obfuscator-LLVM with the following obfuscating transformations: instruction substitution (SUB), bogus control flow (BCF) and control flow flattening (FLA).

In the first test, Obfuscator-LLVM and our system both showed an overhead of $\approx 5\%$. In the second test, the overhead of our system was still $\approx 5\%$, while the overhead of Obfuscator-LLVM was $\approx 13500\%$. Fig. 10 presents the execution times of:

1) the original program,
2) the same program protected using our system,
3) the same program protected using Obfuscator-LLVM with SUB alone,
4) the same program protected using Obfuscator-LLVM with BCF alone,
5) the same program protected using Obfuscator-LLVM with FLA alone,
6) the same program protected using Obfuscator-LLVM with SUB, BCF and FLA.

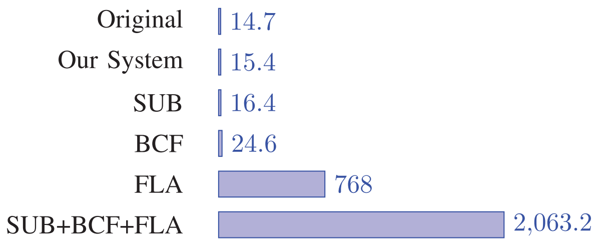


Fig. 10. Execution times in seconds of the original, encrypted and obfuscated versions 7-Zip in the second experiment.

The results are quite expected. Our system is not affected by the contents of the functions, and/or the number of times the function is called as long as the protected code mostly executes in the hypervisor.

C. Randomized Experiment: Lame

For this experiment we used the main executable file of the LAME MP3 encoder [61]. We downloaded the latest LAME source from SourceForge and built a 32-bit version of LAME on Windows 10 Professional x64. We chose a predetermined set S that includes all functions that belong to the `lame` namespace. The set S covers 56% of LAME's main executable functions. For the encryption, we constructed 11 subsets of S: $S_0, S_1, \ldots S_{10} \subseteq S$, where S_i consists of $\frac{i}{10}$ fraction of functions from S selected at random. The encryption resulted in 11 protected executables $P_0, P_1, \ldots, P_{10}$, where P_0 is the original program and P_{10} has all the functions in S encrypted. Each executable was invoked 1000 times in three different configurations (3000 times in total):

1) fixed bit rate 128kbps encoding — default LAME behavior,
2) fixed bit rate jstereo 128kbps encoding, high quality,
3) fast encode, low quality (no psycho-acoustics).

We measured the average execution time for each configuration (1–3) and each protected executable P_i. Fig. 11 depicts the overhead of the encrypted executables in percents.

D. Randomized Experiment: 7-Zip

For this experiment we selected a large subset of 7-Zip functions. Our set S included all the functions of the LZMA algorithm. These functions lay within the "_Lzma" namespace and are prefixed with _Lzma within the source code. We analyzed the decompression time of the latest Linux kernel to date (4.15.6). As in section VII-C, for the encryption, we constructed 11 subsets of S: $S_0, S_1, \ldots S_{10} \subseteq S$, where S_i consists of $\frac{i}{10}$ fraction of functions from S selected at random. The encryption resulted in 11 protected executables $P_0, P_1, \ldots, P_{10}$, where P_0 is the original program and P_{10} has all the functions in S encrypted. Each executable was invoked 1000 times and its average execution time was measured. Fig. 12 depicts the overhead of the encrypted executables in percents.

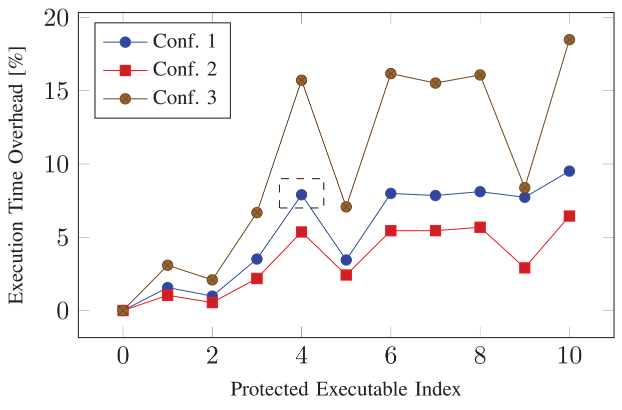


Fig. 11. The overhead of protected executables. Each line represents a single configuration. A mark on a line represents an average execution time overhead (in percents) of a protected executable compared to the original executable. Consider the mark inside the dashed square, which corresponds to P_4 when run in the first configuration. According to this mark, P_4 is slower than the original program (P_0) by 7.9%.

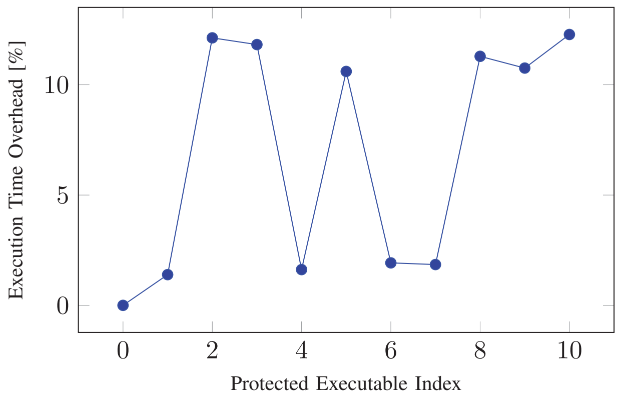


Fig. 12. The overhead of protected executables. A mark on a line represents an average execution time overhead (in percents) of a protected executable compared to the original executable.

E. Built-in Profiler

The built-in profiler allows one to get a better view of the relationships between the encrypted and non-encrypted functions. This information might be critical as every branching to a non-encrypted function requires two costly operations: a VM-entry and a VM-exit. As the number of transitions between encrypted and non-encrypted functions increases, so does the total execution time of the program. The built-in profiler records all the transitions from an encrypted to a non-encrypted function (step 8 in Algorithm 1). For each transition, the hypervisor stores the faulting address (i.e. the address of the non-encrypted function) and the total number of transitions to this address that occurred during the current execution.

The dynamic analysis behavior of the built-in profiler provides a great advantage over static analysis as the control flow of a program frequently depends on its input and cannot be known in advance. However, a combination of the two analysis techniques may be used to ease the profiling process.

We note that it may be undesirable or impossible to encrypt some functions that are suggested by the profiler. Functions residing in shared libraries, that are used by both protected and regular programs, cannot be encrypted. The effect of encrypting a function, that is called from encrypted and non-encrypted functions, is unpredictable, since the transitions overhead from encrypted functions decreases, but the transitions overhead from non-encrypted functions increases. Finally, the decrypted functions are stored in the processor's cache, which has a limited capacity. Therefore, encryption of too many functions can lead to thrashing and performance degradation.

F. Case Study: OpenSSL

In this case study, we analyze an OpenSSL library [62], *libcrypto*, which provides fundamental cryptographic functions for *libssl*. We cloned the latest *openssl* from its original Git repository and built a 32-bit version of *libcrypto* DLL on Windows 10 Professional x64. We used the OpenSSL command line tool in order to invoke functions from *libcrypto* DLL. Specifically, we used its RSA private key generation command, `genrsa`. Each test was executed 1000 times and an average execution time was computed.

We demonstrate how our built-in profiler may be used to improve the performance overhead. At each iteration, we select a set of functions for encryption, run the system and augment the set of selected functions according to the profiler's suggestions. This allows us to improve the performance of the protected program by two orders of magnitude.

For the first iteration, we selected two vital functions, each of which caused hundreds of thousands of branchings (each branching and return requires a VM-entry and a VM-exit). The execution time of the program increased by 573%.

The branchings we have encountered during the profiling process can be divided into three types:

1) a direct call to an internal function,
2) an indirect call to an internal function using the Export Address Table,
3) a direct call to an external function.

Type 1 is the simplest as the function to be called is a real function — it lies within the protected executable and can be referenced by its name in the configuration file. Type 2 represents a function that was generated automatically by the compiler. It does not have a name and therefore must be referenced by its address in the configuration file. Type 3 requires encryption of the external DLL that contains the called function.

For the second iteration, we augmented the set of encrypted functions according to the profiler suggestion. The execution time overhead improved from 573% to 214% of the original's. One should note here that encrypting a function, suggested by the profiler, does not necessarily yield better results initially as it may contain additional branchings (e.g. see the second iteration in Fig. 13).

We performed 12 iterations in total. The last profiling iteration was composed of 59 encrypted functions from which 61% are a result of type 1 branching and 39% are a result of type 2 branching. During our tests we found that encrypting type 3 branching until our last phase gave little benefit. The execution time overhead of the last iteration was 18% of the

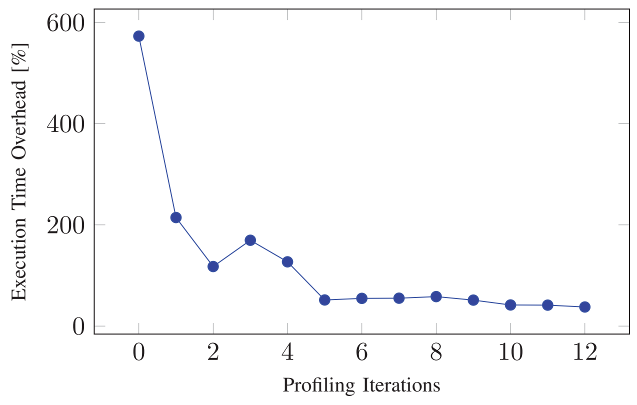


Fig. 13. The overhead of the protected executable during profiling iterations. A mark at position i represents the overhead in percents of the protected executable compared to the original executable at the ith iteration. Note the exponential behavior of the overhead improvement.

	Compression	Decompression
Original	122.7143	6.4979
Protected	122.7701	6.5617

TABLE I
EXECUTION TIME (IN MILLISECONDS) OF 7-ZIP TESTS.

original's. It should be noted that the process could continue further in order to achieve even better results. Fig. 13 depicts the results we obtained.

G. Case Study: 7-Zip

In this case study, we analyze 7-Zip, a file archiver. We downloaded 7-Zip source code from the official website and built a 32-bit version of 7-Zip. Two main functionalities of 7-Zip were tested:

1) adding files to an archive (the "a" command line option),
2) extracting files from an archive (the "e" command line option).

For each of the tests, we selected 3 important functions of 7-Zip that are called during compression and decompression. Afterwards, we downloaded the latest stable Linux kernel (4.15.6) tarball from *kernel.org*. For the first test, we decompressed the tarball and compressed the resultant tar file using the 7-Zip archive option. For the second test, we decompressed the resultant 7z file using the 7-Zip extract option. During the profiling process, we discovered that the compiler in-lined most of the frequently-called functions. Therefore, few profiling iterations were required. In both tests, the overhead of the encrypted program was less than 1%. The exact execution times are presented in Table I.

Section VII-B compares the performance of an encrypted 7-Zip with the performance of an obfuscated 7-Zip. The section concludes that the performance degradation of an encrypted 7-Zip is $\approx$5%. The discrepancy with the result of this section (1%) is due to the profiling iterations that were applied in this case.

H. Case Study: Apache Web Server

In this case study, we analyze the Apache HTTP Server [63]. Specifically, we selected two libraries that are heavily used by Apache. The first library, `libhttpd`, contains, among other things, core HTTP functionallity such as URI parsing and HTTP Request reading. The second library is the Apache Portability Runtime library, `libapr`, which provides consistent interfaces to underlying OS-specific infrastructure (e.g., network sockets). We downloaded the latest (httpd-2.4.34) Apache HTTP Server Unix sources directly from the official Apache website along with all the required dependencies (APR, APR-Util, APR-Iconv, Expat and PCRE). The HTTP daemon, `httpd`, was built in 32-bit Release configuration using the built-in makefile, while all of the other dependencies were built using Microsoft Visual Studio 2015 in 32-bit RelWithDebInfo configuration. As a starting point, we selected three functions, within `libhttpd`, that are heavily used by Apache for HTTP requests handling. As our benchmark utility, we used the Apache Benchmark (`ab`) tool. We measured the time it took for the server to handle 20,000 requests (`-n` option in `ab`). Table II summarizes the benchmark results of the original (unencrypted) application and the encrypted application after 0, 1 and 2 iterations of the profiler.

Case	Handling Time
Original	11.46
Iteration 0	17.7
Iteration 1	15.6
Iteration 2	13.8

TABLE II
EXECUTION TIME (IN SECONDS) OF APACHE TESTS.

VIII. CONCLUSIONS

We have seen that the described system can provide high security guarantees. In most cases, the performance penalty of the system is insignificant; in others, the built-in profiler can improve the performance dramatically.

The described system protects only native code. However, the system can be extended to support managed and interpreted languages by, first, translating programs written in these languages to native code, and then encrypting them.

The implementation described in this paper uses virtualization in order to create an isolated environment. We believe that similar security guarantees and performance can be achieved by using SGX [64] instead of VMX.

REFERENCES

[1] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[2] S. Banescu and A. Pretschner, *A Tutorial on Software Obfuscation*. Elsevier, 01 2017.

[3] B. Anckaert, M. H. Jakubowski, R. Venkatesan, and C. W. Saw, "Runtime protection via dataflow flattening," in *Emerging Security Information, Systems and Technologies, 2009. SECURWARE'09. Third International Conference on*. IEEE, 2009, pp. 242–248.

[4] F. B. Cohen, "Operating system protection through program evolution." *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[5] S. Goldwasser and G. N. Rothblum, "On best-possible obfuscation," in *Theory of Cryptography Conference*. Springer, 2007, pp. 194–213.

[6] S. Bhatkar and R. Sekar, "Data space randomization," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 1–22.

[7] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, Tech. Rep., 2008.

[8] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE, 1997, pp. 67–72.

[9] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[10] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in *International Conference on Information and Communications Security*. Springer, 2004, pp. 187–199.

[11] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*. Springer, 2008, pp. 100–120.

[12] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 189–200.

[13] A. Salem and S. Banescu, "Metadata recovery from obfuscated programs using machine learning," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. ACM, 2016, p. 1.

[14] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 674–691.

[15] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 25–36.

[16] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. De Bosschere, "Towards tamper resistant code encryption: Practice and experience," in *International Conference on Information Security Practice and Experience*. Springer, 2008, pp. 86–100.

[17] J. Bringer, H. Chabanne, and E. Dottax, "White box cryptography: Another attempt." *IACR Cryptology ePrint Archive*, vol. 2006, no. 2006, p. 468, 2006.

[18] M. Karroumi, "Protecting white-box aes with dual ciphers," in *International Conference on Information Security and Cryptology*. Springer, 2010, pp. 278–291.

[19] Y. Xiao and X. Lai, "A secure implementation of white-box aes," in *Computer Science and its Applications, 2009. CSA'09. 2nd International Conference on*. IEEE, 2009, pp. 1–6.

[20] O. Billet, H. Gilbert, and C. Ech-Chatbi, "Cryptanalysis of a white box aes implementation," in *International Workshop on Selected Areas in Cryptography*. Springer, 2004, pp. 227–240.

[21] Y. De Mulder, P. Roelse, and B. Preneel, "Cryptanalysis of the xiao–lai white-box aes implementation," in *International Conference on Selected Areas in Cryptography*. Springer, 2012, pp. 34–49.

[22] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel, "Cryptanalysis of white-box des implementations with arbitrary external encodings," in *International Workshop on Selected Areas in Cryptography*. Springer, 2007, pp. 264–277.

[23] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, and B. De Sutter, "Tightly-coupled self-debugging software protection," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. ACM, 2016, p. 7.

[24] P. Ferrie, "Attacks on more virtual machine emulators," *Symantec Technology Exchange*, vol. 55, 2007.

[25] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, "An efficient vm-based software protection," in *Network and System Security (NSS), 2011 5th International Conference on*, Sept 2011, pp. 121–128.

[26] J. Kinder, "Towards static analysis of virtualization-obfuscated binaries," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 61–70.

[27] R. Rolles, "Unpacking Virtualization Obfuscators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1.

[28] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[29] "Stunnix C/C++ Obfuscator," http://stunnix.com/, 2018, [Online; accessed 25-Feb-2018].

[30] "The Tigress C Diversifier/Obfuscator," http://tigress.cs.arizona.edu/, 2018, [Online; accessed 25-Feb-2018].

[31] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security), Washington, DC*, 2015.

[32] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2007, vol. 3.

[33] "AMD64 Architecture Programmer's Manual Volume 2: System Programming," AMD, 2010.

[34] "ARM architecture reference manual ARMv8-A," ARM Ltd., 2013.

[35] T. Morris, "Trusted platform module," in *Encyclopedia of cryptography and security*. Springer, 2011, pp. 1332–1335.

[36] M. Lipow, "Number of faults per line of code," *IEEE Transactions on software Engineering*, no. 4, pp. 437–439, 1982.

[37] O. Alhazmi, Y. Malaiya, and I. Ray, "Security vulnerabilities in software systems: A quantitative perspective," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2005, pp. 281–294.

[38] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," *CanSecWest/core06*, 2006.

[39] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 335–350.

[40] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 380–395.

[41] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "Hima: A hypervisor-based integrity measurement agent," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 461–470.

[42] J. Rutkowska and R. Wojtczuk, "Qubes os architecture," *Invisible Things Lab Tech Rep*, vol. 54, 2010.

[43] S. T. King and P. M. Chen, "Subvirt: Implementing malware with virtual machines," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 14–pp.

[44] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[45] PELock LLC. PELock: Software Protection. [Online]. Available: https://www.pelock.com/

[46] The UPX Team. UPX: the Ultimate Packer for eXecutables. [Online]. Available: https://upx.github.io/

[47] J. Raber, "Columbo: High perfomance unpacking," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 507–510.

[48] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177.

[49] VMware. VMware Workstation Pro. [Online]. Available: https://www.vmware.com/il/products/workstation-pro.html

[50] Oracle. VirtualBox. [Online]. Available: https://www.virtualbox.org/

[51] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 143–158.

[52] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "Bitvisor: A thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 121–130.

[53] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Science of Computer Programming*, vol. 74, no. 1-2, pp. 13–22, 2008.

[54] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[55] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. OHanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.

[56] S. Andersen and V. Abella, “Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies,” 2004.

[57] R. E. Kessler and M. D. Hill, “Page placement algorithms for large real-indexed caches,” *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 338–359, 1992.

[58] X. Zhang, S. Dwarkadas, and K. Shen, “Towards practical page coloring-based multicore cache management,” in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 89–102.

[59] Intel. 8th Generation Intel Core i7 Processors. [Online]. Available: https://ark.intel.com/products/series/122593/8th-Generation-Intel-Core-i7-Processors

[60] C. Lambrinoudakis, G. Pernul, and M. Tjoa, *Trust, Privacy and Security in Digital Business: 4th International Conference, TrustBus 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007. [Online]. Available: https://books.google.co.il/books?id=ci-rNuWTLa4C

[61] “LAME,” http://lame.sourceforge.net/, 2018, [Online; accessed 25-Feb-2018].

[62] OpenSSL Software Foundation, “OpenSSL: Cryptography and SSL/TLS Toolkit,” https://www.openssl.org/, 2018, [Online; accessed 25-Feb-2018].

[63] Apache. The Apache HTTP Server Project. [Online]. Available: https://httpd.apache.org/

[64] V. Costan and S. Devadas, “Intel SGX Explained.” *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

PIV

PREVENTING EXECUTION OF UNAUTHORIZED NATIVE-CODE SOFTWARE

by

A. Resh, M. Kiperberg, R. Leon, N.J. Zaidenberg 2017

International Journal of Digital Content Technology and its Applications (JDCTA), Vol. 11, No. 3, pp. 72-90

http://www.globalcis.org/jdcta/ppl/JDCTA3804PPL.pdf

Preventing Execution of Unauthorized Native-Code Software

[1]Amit Resh, [2]Michael Kiperberg, [3]Roee Leon, [4]Nezer J. Zaidenberg
[1] *Deparment of Mathematical IT, University of Jyväskylä, Finland, amitr44@gmail.com*
[2] *Faculty of Sciences, Holon Institute of Technology, Israel, mkiperberg@gmail.com*
[3] *Deparment of Mathematical IT, University of Jyväskylä, Finland, roee.leonn@gmail.com*
[4] *School of Computer Sciences, College of Management, Israel, nzaidenberg@me.com*

Abstract

The business world is exhibiting a growing dependency on computer systems, their operations and the databases they contain. Unfortunately, it also suffers from an ever growing recurrence of malicious software attacks. Malicious attack vectors are diverse and the computer-security industry is producing an abundance of behavioral-pattern detections to combat the phenomenon. This paper proposes an alternative approach, based on the implementation of an attested, and thus trusted, thin-hypervisor. Secondary level address translation tables, governed and fully controlled by the hypervisor, are configured in order to assure that only pre-whitelisted instructions can be executed in the system. This methodology provides resistance to most APT attack vectors, including those based on zero-day vulnerabilities that may slip under behavioral-pattern radars.

Keywords: *Hypervisor, Trusted computing, Whitelisting, Attestation, APT-protection, Cyber-security*

1. Introduction

An abundance of malicious software attacks plague the computer software industry. The attack methodologies are diverse, ranging from code-injection, buffer-overflow, viruses, worms and Trojans to rootkits. Malicious code is usually designed to gain access to and steal the victim's data, such as personal information, credentials, trade secrets, or to gain access to the victim's system in order to take advantage of the resource for inflicting further damage. Malicious code motivation is predominantly financial but in some case other motivations may exist as well.

In many cases malicious attacks are not carried out in a single shot. Many attacks are multi-faceted, containing several intermediate steps, each designed to progress the offender to the next level of penetration before reaching the final goal. As an example, [1] details 5 stages of a Web malware attack leading from entry to execution on the compromised system:

- Entry – malicious code enters the victim system as a result of a drive-by download occurring when visiting a hacked site or following a malicious link in an email.
- Traffic Distribution – drive-by downloads execute inside browsers. Their primary goal is to download an exploit kit. Traffic redirection occurs to conceal the IP address from which the exploit kits are eventually downloaded.
- Exploits – once an exploit kit is downloaded it attempts to locate a system vulnerability that it can exploit in order to progress the attack. Exploits are usually encapsulated in PDF, FLASH, Java, JS or HTML files.
- Infection – once a vulnerability is found by the exploit kit, it is used to download the actual malware's executable code. SophosLabs identify several common malware payloads: Zbot(Zues) – steals personal information by logging keystrokes and grabbing display frames; Ransomeware – restricting access to the user's resources and demanding payment to restore access; PWS – steals user credentials and allows remote access; Sinowal(Torpig) – installs a rootkit to steal credentials and allow remote access; FakeAV – a Fake antivirus that "finds" fake viruses and demands payment to "clean" them out.
- Execution – the downloaded malware has been installed in the victim system and is executed. This is the stage where the actual damage is inflicted.

Other types of attacks exist as well, each seeking to abuse system or human vulnerabilities in order to inflict damages, gain access to privileged information or completely take control. Many of these attacks are similarly multi-stage. Attacks may exploit all or some of the following common stages:

- Entry – malicious code enters the system as a result of a malicious email attachment, a bogus executable installation a buffer-overflow, a USB disk insertion, a worm or a virus spreading.
- Non-privileged execution – in this mode of execution, malicious code that has entered the system executes in a low privileged level. It may still inflict some damage, however that damage is usually limited and may eliminate its capability to achieve persistency. In that case, the malicious code will disappear when the system is rebooted.
- Escalation: privileged execution – a much more hazardous case occurs when an unprivileged code exploits a system vulnerability (usually in the OS) and manages to escalate its privilege. It is beyond the scope of this text to describe the mechanisms that may be employed to achieve this, but the statistics are most staggering. Malicious code that gains privileged access may freely write to the filesystem on disk, to the main memory – both to user and to OS space, to the system registry or even to the boot record or BIOS memory.
- Acquiring Persistency – using the capabilities of privileged execution, malicious code can strive for persistency. In other words, the capability to survive system reboot as well as a complete system power-cycle. Achieving this level is the first step in "securing" the malicious code's survival in the compromised system. Many infections will also go to great lengths to camouflage their existence using a variety of methods, some very cunning, to avoid detection and removal.
- Compromised system – once malicious code has persistent execution on the system the perpetrator can potentially steal sensitive data, log keyboard activity to steal messages or passwords, grab screenshots or even achieve full remote-control of the system.

While system penetration is possible to some extent, without resorting to execution of unrecognized instructions in the system – ultimately all penetration goals are served only by executing some form of (rogue) executable instructions, which were not part of the system before the penetration. The methodology proposed by the authors in this paper, takes advantage of this fact, to provide an efficient way to protect against most such invasions, performed by a large variety of penetration techniques and also in many cases that utilize a previously unknown zero-day vulnerability.

The authors propose an approach whereby native-code is verified just before it receives execution rights. To achieve this, the entire system is first "whitelisted" by generating a database that contains signatures for every executable code-page that exists in the system's executable files, DLLs, drivers etc. A hypervisor is utilized to intercept and verify every execution attempt, at a page granularity, according to the whitelist database. The system is based on the approach proposed by Averbuch et al. [2] [3], in which an attested kernel module is responsible for performing cryptographic operations.

Hypervisors have been previously used to secure systems. For example, the Software-Privacy Preserving Platform (SP^3) [4] utilizes a hypervisor to maintain isolated memory-pages in ***protection-domains***. Physical pages in the system can be individually encrypted with a symmetric-key, where each domain has an associated set of keys whose pages it is allowed to use. The hypervisor intercepts interrupts and exceptions and uses shadow page-tables to manage decryption and encryption of the appropriate pages when the application shifts between domains. This methodology keeps domain access to protected pages isolated from other domains as well as from the OS. The hypervisor stores the key-database and domain key-associations in its own isolated memory. We have previously extended Truly-Protect hypervisor to support digital rights protection for digital video distribution. [5] [6] This is our second extension of TrulyProtect Hypervisor.

2. Thin hypervisor

A hypervisor, also referred to as a Virtual Machine Monitor (VMM), is software, which may be hardware assisted, to manage multiple virtual machines on a single system. The hypervisor virtualizes the hardware environment in a way that allows several virtual machines, running under its supervision, to operate in parallel over the same physical hardware platform, without obstructing or impeding each other. Each virtual machine has the illusion that it is running, unaccompanied, on the entire hardware platform. The hypervisor is referred to as the Host, while the virtual machines are referred to as Guests.

Hypervisors have been in use as early as the `60s on IBM mainframe computers [7]. After 2005 Intel and AMD introduced hardware support for virtualization (Intel VT-X [8], AMD AMD-V [9]) which allows implementing hypervisors on the ubiquitous PC platforms.

In order to support multiple OS guests, a hypervisor must unobtrusively intercept OS access to hardware resources so it can attend to them itself. The hypervisor can then manage hardware allocations that maintain proper separation between the Guests. The Guest OS is unaware of the hypervisor's intervention, as it experiences a normal hardware access cycle. The only distinction being the elapsed time, since the hypervisor mediation has a time-toll.

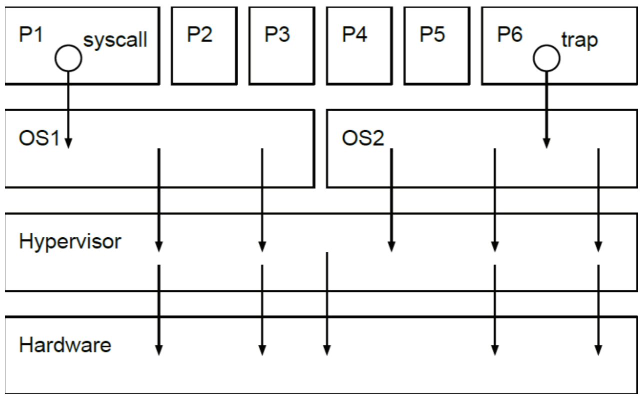


Figure 1. Virtualized system featuring a hypervisor and two operating systems executing 6 programs. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions and other interrupts, transfer control from user mode applications to their operating system. The operating system handles these conditions by requesting some service from the underlying hardware. The hypervisor intercepts those requests and handles them according to some policy.

To intercept OS hardware access, hypervisors can be configured to intercept privileged instructions, memory access, interrupts, exceptions and I/O, which are the OS vehicles for hardware access. Executing an intercepted privileged instruction causes a hypervisor VM_EXIT. In other words, the Guest is exited and the configured hypervisor intercept-routine is executed. When this occurs, the CPU mode changes from Guest-mode to Host-mode.

Guest applications that require hardware resources, execute system calls to request support from their OS. Figure 1 depicts this chain-of-execution for a hypervisor with two Guest stacks. After fulfilling the intercept, the hypervisor indiscernibly returns to the Guest. While hypervisors were generally designed to serve as virtual machine monitors, hypervisors, which control the underlying hardware platform, are also very good platforms to serve as software security facilitators.

The authors propose to use a hypervisor environment for securing a single Guest stack. Rather than wholly virtualizing the hardware platform, a special breed of hypervisor, called a ***thin-hypervisor***, is used [10] [11]. The thin-hypervisor is configured to intercept only a small portion of the system's privileged events. All other privileged instructions are executed without interception, directly, by the OS. The thin-hypervisor only intercepts the set of privileged instructions that allows it to protect an internal secret (such as cryptographic key material) and protect itself from subversion. Figure 2 depicts a thin-hypervisor supporting a single Guest stack. Since the thin-hypervisor does not control most of the OS interaction with the hardware, multiple OSs are not supported. However, system performance is kept at an optimum.

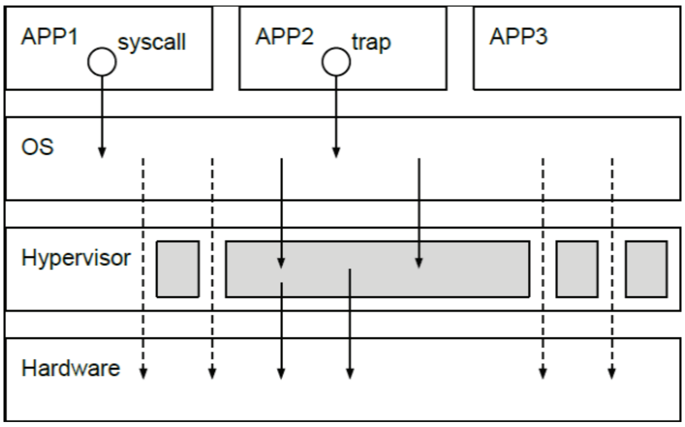


Figure 2. Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service from the underlying hardware. A thin hypervisor can intercept some of those requests and handle them according to some policy.

Thin hypervisors have been previously used for security purposes. TrustVisor [12] is a thin hypervisor that enables isolated execution of designated portions of an application. TrustVisor is booted securely by making use of a TPM chip and once in operation, it depends on hardware virtualization to isolate portions of memory with Secondary Level Address Translation (SLAT) as well as protect memory from DMA access by physical devices with DEV or IOMMU. TrustVisor utilizes this capability to (i) protect itself; and (ii) extend TPM facilities to a so-called µTPM environment that is used to provide high-speed trusted-computing primitives. These capabilities are further used by TrustVisor to achieve its ultimate goal of supporting a totally-isolated execution environment for designated self-contained software routines, called PALs (Pieces of Application Code). Software developers designate the portions of their codes that require isolation and group them into appropriate PALs. The developers register the PALs by providing a description of PAL bounds as well as memory regions they need to access. The TrustVisor guarantees that when PALs are called they operate in an isolated memory environment until they are exited.

A thin-hypervisor facilitates a secure environment by:

1. Setting aside portions of memory that can be accessed only when the CPU is in Host mode
2. Storing cryptographic key material in privileged registers and
3. Intercepting privileged instructions that may compromise its protected memory or key material

A thin-hypervisor is less susceptible to being hacked as a result of vulnerabilities, since its code and complexity are greatly reduced, as compared to a full-blown hypervisor.

Once this environment is correctly setup and configured, the thin-hypervisor can be utilized to carry out specific operations, which may include use of the internally stored key material, in a protected region of memory. As a result of the tightly configured intercepts and absolute host control of select memory regions, this activity can be guaranteed to protect both the secret key material and the operations' results.

The thin-hypervisor can effectively protect the secret key-material, after it is safely stored in privileged registers and the thin-hypervisor is correctly configured and active. However, the procedure by-which the secret material gets stored while the thin-hypervisor is being setup – is delicate business, since an adversary can potentially grab the secret at that point. An additional question, requiring an answer, is where the secret is kept while the thin-hypervisor is not active?

The authors' approach to solving these issues is based on an approach described in [13] and is comprised of the following principles:

- While the thin-hypervisor is not active, the secret key material shall not be stored anywhere in the system
- When setting up a thin-hypervisor, an external system shall be used to verify that the thin-hypervisor has control over the underlying hardware
- The same external system that verifies the thin-hypervisor shall provide the secret key-material

The first principle is important to rule out the possibility of keeping secret material under the cover of obfuscation, which is known to be ultimately vulnerable. The second and third principles require maintaining a remote key-server system and equipping it with the facilities to verify that a thin-hypervisor on a remote system has been properly setup and configured, such that a trusted environment is primed and can accept secret material.

2.1 Adversary Model

We assume that an adversary is freely able to access system memory for writing and reading. Memory can be accessed for writing in a variety of ways. For example, contents can be loaded from disk, arrive over a communication channel or be injected directly into memory by an executing application. We further assume that an adversary is also able to write to some memory regions that should in principle be protected by the OS, based on exploiting system vulnerabilities. Such regions include, but are not limited to, application code, privileged kernel-mode code and system drivers. Accordingly, memory that has been accessed for writing, by the application or by the OS, is never trusted for execution purposes.

Furthermore, it is assumed that an adversary cannot obstruct the operation of a root (primary) hypervisor that is based on hardware virtualization, as well as secondary memory translation (i.e., EPT) and IOMMU that operate at a privilege that is higher than the OS when a hypervisor is active.

Adversary attacks that are based on manipulating pure data in memory, in such a way as to render legitimate code malicious (referred to as code-reuse) are not considered.

2.2 Contribution

The authors propose a methodology and system that achieve a strong system-wide protection against execution of a wide array of unauthorized code penetrations. Our approach is distinguished from previous efforts by the implementation of an attested thin-hypervisor, which launches in an existing OS and which extends its security model over existing legacy applications without requiring their modification.

The unique approach described here allows a system to dynamically shift between protected and unprotected modes of operation. This situation can be appreciated, for example, in a BYOD situation, where enterprise employees can use their own computers for private (unsecure use) without enduring the performance overhead associated with protection, then shifting dynamically into protected mode to run office applications that require tight security. Applications that execute in protected mode, shall be protected and isolated from malicious code the computer may have contracted.

Dynamically shifting into protected mode is based on the capability to activate a thin-hypervisor after an OS already prevails. Securing trust in this situation entails administering a remote attestation procedure to establish a trusted environment in an otherwise untrusted computer system.

3. Achieving trust in a remote system

The problem of remote software authentication, determining whether a remote computer system is running the correct version of a software, is well known [14] [15] [16] [17] [18] [19]. Equipped with a remote authentication method, a service provider can prevent an unauthenticated remote software from obtaining some secret information or some privileged service. For example, only authenticated gaming consoles can be allowed to connect to the gaming networks [20] [21] [22] and only authenticated bank terminals can be allowed to fetch records from the bank database [23].

The research in this area can be divided into two major branches: hardware assisted authentication [24] [25] [26] and software-only authentication [14] [15] [27]. In this paper we concentrate on software-only authentication, although the system can be adapted to other authentication methods, as well. The authentication entails simultaneously authenticating some software component(s) or memory region, as well as verifying that the remote machine is not running in virtual or emulation mode. Software-only authentication methods may also involve a challenge code that is sent by the authentication authority, and executed on the remote system. The challenge code computes a result that is then transmitted back to the authority. The authority deems the entity to be authenticated if the result is correct and was received within a predefined time-frame. The underlying assumption, which is shared by all such authentication methods, is that only an authentic system can compute the correct result within the predefined time-frame. The methods differ in the means by which (and if) they satisfy this underlying assumption.

Kennell and Jamieson proposed [14] a method that produces the result by computing a cryptographic hash of a specified memory region. Any computation on a complex instruction set architecture (Pentium in this case) produces side effects. These side effects are incorporated into the result after each iteration of the hashing function. Therefore, an adversary, trying to compute the correct result on a non-authentic system, would be forced to build a complete emulator for the instruction set architecture to compute the correct side effects of every instruction. Since such an emulator performs tens and hundreds of native instructions for every simulated instruction, Kennell and Jamieson conclude that it will not be able to compute the correct result within the predefined time-frame. The method of Kennel and Jamieson was further adapted, by the authors, to modern processors [13]. The adaptation solves the security issues that arise from the availability of virtualization extensions and multiplicity of execution units.

Establishing a thin-hypervisor that receives a remote secret (cryptographic key) in confidence and which may execute cryptographic operations with that secret key, provides an excellent software-only platform to utilize and sustain trust. The utilization of trust is based on being able to deliver encrypted or cryptographically-signed material to the remote system. The thin-hypervisor can decrypt and/or validate the received material and act accordingly. Any attempts to make changes, additions or deletions to the delivered material will inevitably be detected by the thin-hypervisor, provided the secret key is kept secret. Trust sustainability is upheld by eliminating any possible access to the secret material as well as rejecting any attempts to disrupt the code or state of the thin-hypervisor. Fortunately, a hypervisor has the available facilities to achieve just that.

Setting up a trusted thin-hypervisor on a remote system, while adhering to the 3 principles noted in the previous section, involves the following validations:

1. The thin-hypervisor's code is validated
2. The validated code is the one that executes when a VM_EXIT occurs
3. The thin-hypervisor controls the underlying hardware

3.1 Overview of the methodology

The vehicle to perform this remote verification is a piece of code, called an attestation-challenge [28] [29]. The attestation-challenge is administered by the key-server to the remote machine, as it is configuring the thin-hypervisor. The remote machine is required to load and execute the challenge code, returning an attestation result to the key-server within a pre-limited time-frame. The attestation-challenge calculates the checksum of the thin-hypervisor code, but in addition convolutes the checksum calculation with hardware side-effects, sampled by the challenge as it is executing. The side-effect samples are hardware-registers that count hardware events, such as cache hits or misses, TLB hits or misses etc.

The key-server considers a correct response received within the allotted time-frame, proof that the correct thin-hypervisor code is executing and it has true control of the remote system's hardware. Upon receiving a correct response the attestation can provide keys that the attested computer hides and protects in its CPU. [30]

3.2 Remote attestation

As described above, the attestation challenge calculates the checksum of the thin-hypervisor's code convoluted by hardware event samples. The attestation challenge is composed of several computational nodes. Each node executes a single operation related to the challenge result calculation and then branches to the next node according to the current result value. Three different branches are possible for each node:

- Branch A: if the result parity is even (50% chance)
- Branch B: if the sign bit is set (25% chance)

- Branch C: Otherwise (25% chance)

Branch target nodes may be the same or different, for each possible branch option. The variety of nodes include:

- Checksum operation – Sum a hypervisor code value
- XOR hardware counter – Xor hardware-event-counter *i* with current checksum result
- AND hardware counter – AND hardware-event-counter *i* with current checksum result
- Multiply hardware counter – Multiply hardware-event-counter *i* with current checksum result
- MAC calculation (such as SHA-1)

where *i* is a Data-Cache Hit, Data-Cache Miss, TLB-Hit, TLB Miss, etc. Due to the multiple branches stemming from each node, the entire set of nodes comprises a network.

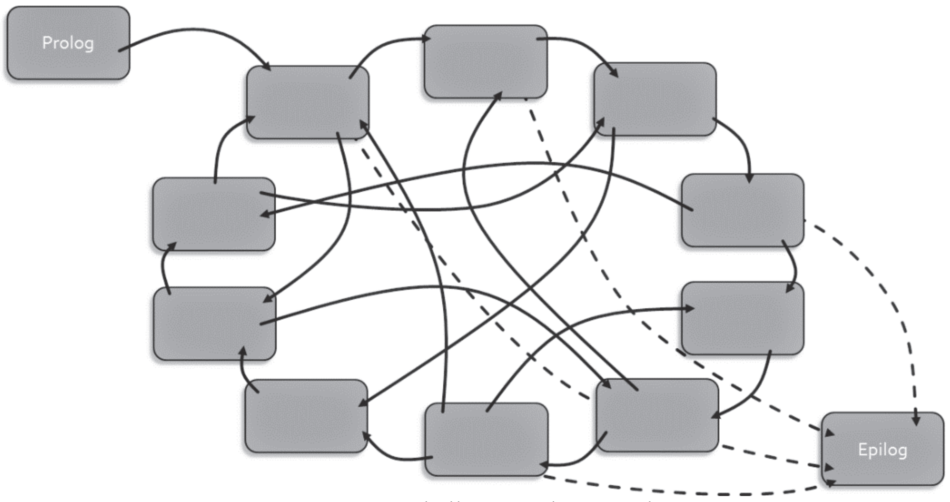


Figure 3. A challenge node network.

The node network is built to guarantee that every circuit contains at least one of each node-type. The first node to execute is the "Prolog" node, which sets up the environment and configures the hardware side-effect counters. The "Epilog" node is the last node to execute. It performs clean-up and returns the final challenge result.

Checksum calculation is performed by summing a wide virtual space that is redundantly mapped to the physical memory space that contains the code regions need to be attested along with their page-tables. The challenge is always accompanied by a (pseudo-random) *virtual map* that is designed to map the relatively small physical-page region to the relatively large virtual space. Naturally, each physical-page is mapped to multiple virtual-pages. The physical-page region includes:

- The thin-hypervisor code pages
- The challenge code page (all the code of the nodes)
- The page-table pages that define the virtual map

The challenge nodes are contained in a single physical-page, however, individual nodes are mapped at different virtual space locations and as such, each Node executes from a different location.

The checksum calculation order is governed by a pseudo-random-walk according to an LFSR (Linear-Feedback-Shift-Register) generator [31]. Every virtual-space address is visited once, however, physical addresses are visited multiple times. This is designed to induce side-effects. In a check-summing node, the value at each address is accumulated to the checksum. Other node types perform additional action on the current result, such as adding in hardware event counter values or calculating a MAC.

The virtual-space random walk creates pseudo-random data-cache patterns that affect future cache hit/miss events. Similarly, execution of nodes, each at a different virtual location, creates pseudo-random code-cache and TLB cache patterns. Each affecting its corresponding cache hit/miss events. Hardware side-effect convoluting type nodes, incorporate a transient hardware counter result into the accumulated checksum. Thereby, both changing the current result value, as well as node progress flow.

It is stipulated that challenge results calculated in an environment that is different than the intended (for example at attempt to execute our thin-hypervisor under an emulator or as a nested-hypervisor) will generate a significantly different challenge result and thus be easily detected. The possibility of calculating a correct result by means of emulation shall also be impossible within the allotted timeframe restriction. We previously shown the challenge server can generate and manage challenges effectively [32]

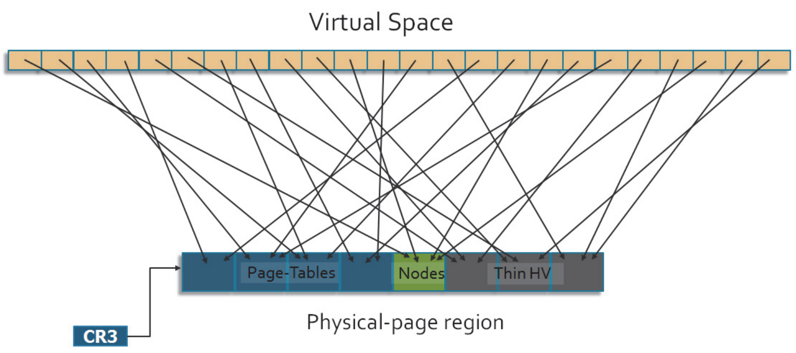


Figure 4. A challenge node network.

4. Controlled execution

4.1 Introduction

The x86 architecture allows the operating system to control memory access rights of applications through the virtual paging mechanism. Similarly, virtualization extensions, which were introduced by Intel and AMD, allow a hypervisor to control memory access rights of operating systems through a mechanism called Second Level Address Translation (SLAT). Intel and AMD refer to this mechanism as Extended Page Table (EPT) [8] and Rapid Virtualization Indexing (RVI) [9], respectively. Virtual paging and SLAT can be used to specify the "read", "write" and "execute" rights of a particular memory page ("execute" rights are controlled by the "NX bit" in virtual paging [8] Unlike virtual paging, SLAT defines the memory access rights of the physical rather than the virtual pages, thus providing the hypervisor with complete control over the access rights in all memory modes.

Our hypervisor uses SLAT to prevent execution of unauthorized software. Initially, the hypervisor forfeits the "execution" rights of all pages, thus effectively intercepting any execution attempt. Upon such intercept, the hypervisor verifies the executing page authenticity, by hashing the page content and comparing it to a precomputed value. After authenticity is established, the hypervisor grants the page "execution" rights but forfeits its "write" rights, thus intercepting attempts to modify authenticated pages. Upon interception of such a modification attempt, the hypervisor grants the page "write" rights but forfeits its "execution" rights. Therefore, at all times, a page can have either "execution" rights or "write" rights, but not both.

Page authentication in its simplest form consists of two steps: hashing and comparison. In the first step, the hypervisor applies a hash function to the page being authenticated. In the second step, the hypervisor checks whether the result of the hash function appears in a database of valid hash values. This database is built ahead of time by scanning the hard drive for installed applications, computing the hash values of the applications' code pages, storing the hash values in a database, and finally signing the database, in order to prevent its unauthorized modification. Section 4.2 contains a detailed description of the database structure.

In some cases, after loading a page into memory, the operating system alters the page's content according to a set of rules called relocations. A relocation describes an absolute address that is referenced by the application that might need to be adjusted. This adjustment is necessary only if the application was loaded to a non-preferred location, but this is usually the case [33] [34]. In order to apply a relocation at offset x, the operating system first computes the relocation offset, which is the difference between the application's actual and preferred loading locations, and then adds this difference to the address at offset x. Conceptually, during a page's authentication, the hypervisor first restores the original values at the relocation offsets, and then computes the hash of the resulting page. In practice, the page is not modified during authentication; instead, the hashing calculation is performed on some temporal value at relocation offsets.

Unfortunately, some pages contain both code and data. Obviously, the hypervisor cannot fully authenticate such pages. On the one hand, granting these pages with "execution" rights will allow execution of any code in the unverified (data) area of the page, and therefore compromise the security of the entire system. On the other hand, the authentic code cannot be executed from a page without "execution" rights. We propose the following solution to this problem. The hypervisor grants the page with "execution" rights but starts monitoring the guest's instruction pointer. Whenever the instruction pointer exits the authenticated area, the hypervisor forfeits the "execution" rights of the page. Section 4.4 contains a detailed description of this process.

The hypervisor monitors the instruction pointer using the processor's debugging facilities. Specifically, the hypervisor resumes the guest in a single-step execution mode. In this mode, the processor generates an interrupt after every executed instruction, thus enabling the hypervisor to verify that only the authenticated portions of the page are executed, and thus maintain appropriate rights for partially authenticated pages. Some processors provide an extension to the single-step mode, in which the interrupt is generated only after execution of branch instructions, such as jumps, calls and returns. The instruction pointer can exit the authenticated area not only due to a branch instruction but also by falling through the last instruction. The hypervisor intercepts the latter case by installing a hardware breakpoint at the byte following the last instruction of the authenticated area.

4.2 Database structure

We begin our detailed explanation of the execution prevention mechanism, by describing the structure of the database that contains the hash values (see Figure 5). That database consists of modules descriptors. Each module descriptor contains information of a specific executable (PE file in Windows [35] or ELF file in Linux [36]) which resides on the machine. Each descriptor is signed by an RSA signature in order to prevent an attacker from manipulating its contents. We note that an attacker can potentially remove module descriptors, but he cannot alter existing descriptors or add new ones. Each module descriptor contains its size, which allows to move to the next descriptor. The descriptor also holds the path of the executable which is represented by this descriptor. The driver uses the path field to identify the descriptor corresponding to the loaded image. As was explained in section 4.1 the verification procedure needs to know the executable's expected location in memory. This information is stored in the "Base" field of the module descriptor.

Finally, the module descriptor contains a list of section descriptors. Each section descriptor corresponds to an executable section of the executable, and contains the following fields:

- Record size – the size of this section descriptor. This field allows to move to the next descriptor.
- Offset – the offset of the section described by this descriptor from the beginning of the image file.
- Length – the size of the section described by this descriptor.
- Page[i] – page descriptor that corresponds to the i^{th} page of the section.
- # Relocs – the amount of relocation descriptors that follow.
- Reloc[i] – relocation descriptor – explained below.
- # Datums – the amount of the datum descriptors that follow.
- Datum[i] – datum descriptor – explained below.

The amount of page descriptors can be deduced as follows. Let L denote the section's offset rounded down to a page boundary and let R denote the sum of section's offset and section's length rounded up to a page boundary. Then the amount of page descriptors if $(R-L)/4096$. In other words, that database holds a page descriptor even for partial pages, i.e. pages that only partially belong to the section. In that case only the bytes that belong to the section are hashed.

The page descriptor consists of the hash value of the corresponding page (or its part), and two indexes to the Reloc[] array: the index of the first relocation and the index of the last relocation that apply to this page. The relocation descriptor consists of two fields: type – which determines whether the relocation applies to an 8-byte or a 4-byte region, and offset – the location in page where the relocation applies. The datum descriptor consists of two fields: offset – offset from the module beginning, value – 8 bytes at that location. The verification procedure uses the datum descriptor array (in addition to the relocation array) during verification of pages that contain relocations that cross page boundaries.

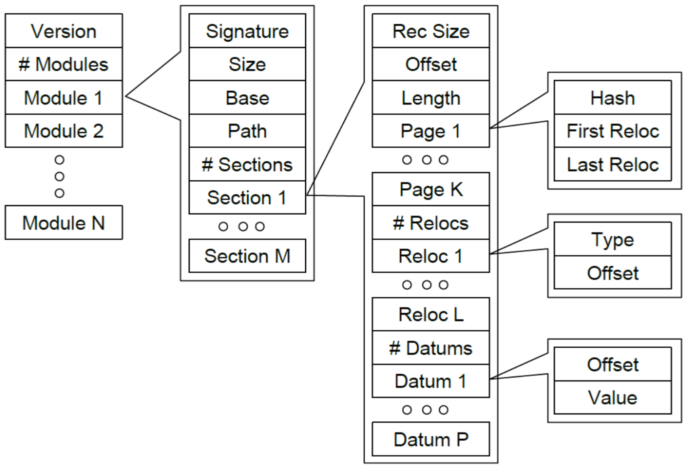


Figure 5. Structure of the database containing the hash values. The database consists of many modules, each of which consists of many sections. Each section contains the hash values of pages that it occupies, the relocations in those pages and datums – values of relocation that cross page boundaries.

4.3 Execution prevention

The hypervisor is part of a device driver, which acts as a mediator between the hypervisor and the operating system. In particular, the driver constructs some data structures that are later used by the hypervisor. We note that the hypervisor cannot (and does not) trust these data structures and therefore their critical parts contain a signature proving their authenticity. During initialization, the driver loads the database containing the hash values to a pageable region of memory, and installs two callbacks; the first callback is invoked when the operating system loads an executable to memory, the second callback is invoked when a process terminates. Both callbacks update a data structure that represents the memory layout of all the processes that are currently active. The data structure is a list of process descriptors. Each process descriptor contains the corresponding process identifier and a pointer to a list of module descriptors. Each module descriptor contains the location in memory of the corresponding module and the database index of this module's descriptor. Figure 6 depicts this data structure.

During the driver's initialization it installs the hypervisor which manages the access rights of physical pages. The hypervisor and the driver callbacks operate concurrently: the callbacks update the memory layout data structure that is used by the hypervisor. Unfortunately, the driver initialization order is determined by the operating system and cannot be affected. Therefore, the operating system may load and initialize some drivers prior to our driver initialization. Consequently, the callback, which is installed during initialization, will not be called on those drivers. Our driver solves the problem, by traversing operating system-specific data structures that contain information about the drivers that were loaded. Figure 7 presents the data structures that are used by a 64-bit version of Windows 8.

Initially the hypervisor forfeits the "execution" rights of all the physical pages. An attempt to execute an instruction triggers an "EPT Violation" (unauthorized access to physical memory) which passes the control to the hypervisor. The hypervisor verifies the authenticity of the page containing the instruction and changes its access rights to "read" and "execute". An attempt to write to this page triggers an "EPT violation" and the hypervisor changes the access rights to "read" and "write". This process is depicted in Figure 8. A detailed description of the verification procedure appears below.

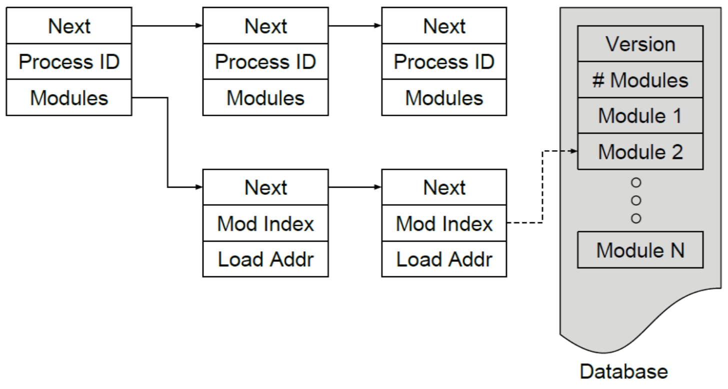


Figure 6. Memory layout data structure. The memory layout consists of a list of process descriptors. Each process descriptor contains the process identifier of the corresponding process and a pointer to a list of module descriptors. Each element of the module descriptors list contains the index of the corresponding module and its location in memory.

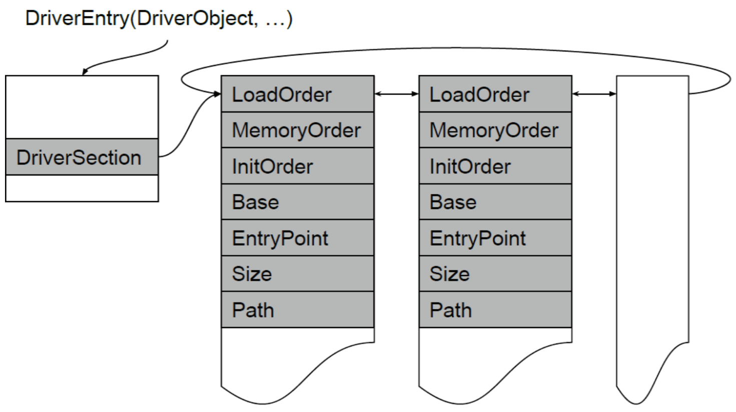


Figure 7. Memory layout data structure. The memory layout consists of a list of process descriptors. Each process descriptor contains the process identifier of the corresponding process and a pointer to a list of module descriptors. Each element of the module descriptors list contains the index of the corresponding module and its location in memory.

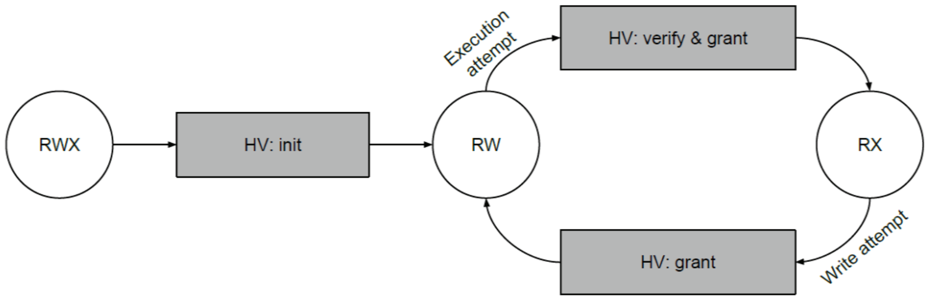


Figure 8. Physical pages access rights state diagram. "RWX" represents full access rights. "RW" represents "read" and "write" access rights. "RX" represents "read" and "execute" access rights.

On a multiprocessor system the hypervisor has a different configuration structure for each processor. In particular, each processor has its own EPT hierarchy, which can independently (of other processors) specify the access rights for each physical page. The hypervisor has to maintain identical configurations of all the EPT hierarchies (with a few exceptions, as we will see later) in order to prevent execution of unauthorized instructions.

Consider the following scenario: an authentic page request execution rights on processor A. The hypervisor verifies the page and grants it "read" and "execute" access rights, thus preventing its further modifications. However, processor B still has "read" and "write" access rights to this page, which enable it to modify the contents of this page. A malicious user can write malicious code to this page using processor B and then execute this malicious code on processor A.

Unfortunately, a processor can modify only its own EPT hierarchies [8]. Therefore, whenever the hypervisor on some processor decides to change the access rights of a page, it sends a request to hypervisors on other processors to make the intended change in their EPT. Only when all the EPT hierarchies of all the other processors were changed, the same change is made on the EPT hierarchy of the initial processor.

The request mechanism is implemented as follows. During its initialization the hypervisor allocates a constant-size queue of requests for each processor, which represents the access rights requests that the hypervisor running on that processor needs to serve. In addition the hypervisor installs an interrupt service routine on a special vector (0xFE), which is not in use by the operating system. The interrupt service routine issues a hypercall with a special value, which informs the hypervisor that its requests queue is not empty. The hypervisor serves this hypercall by applying all the changes described by the requests in the queue and clears the queue. In order to issue a request to another (remote) processor, the hypervisor performs two steps: (1) it inserts a new element to the requests queue of the remote processor, and (2) sends an IPI to the remote processor on the special vector (0xFE). After issuing the request, the hypervisor waits for the changes to be applied. Figure 9 depicts the entire process of access rights modification as it is performed on a multiprocessor system.

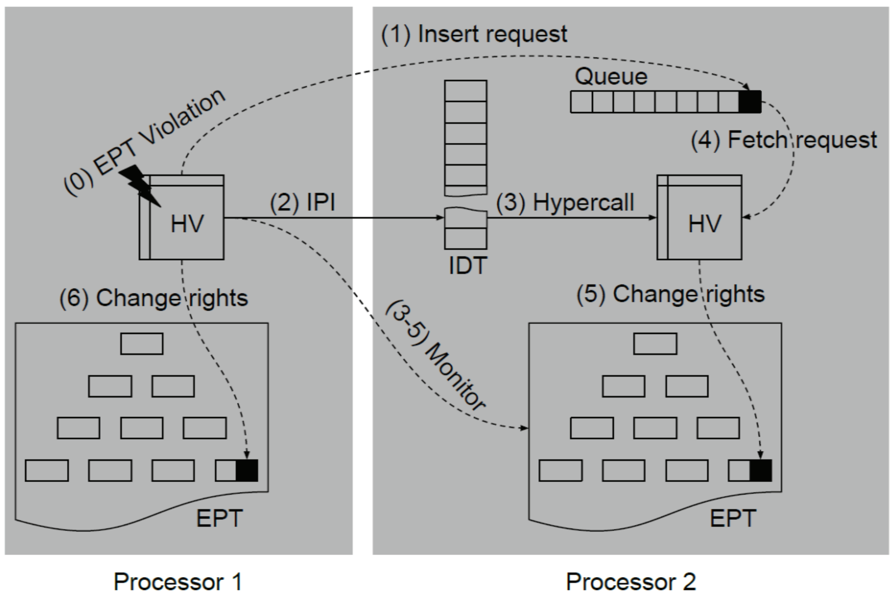


Figure 9. Access rights modification on a multiprocessor system: (0) an EPT violation on processor 1 triggers the hypervisor; (1) the hypervisor inserts a request into the request queue of processor 2; (2) the hypervisor sends an IPI to processor 2; (3-5) the hypervisor monitors the EPT hierarchy of processor 2 and waits for the change to occur; (3) the IPI that was sent in step 2 triggers an ISV; (4) the ISV hypercall to the processor 2 hypervisor; (5) the hypervisor fetches the request and changes the EPT hierarchy accordingly; (6) the processor 1 hypervisor observes that modification in the remote EPT hierarchy and performs the same modification in its local EPT hierarchy.

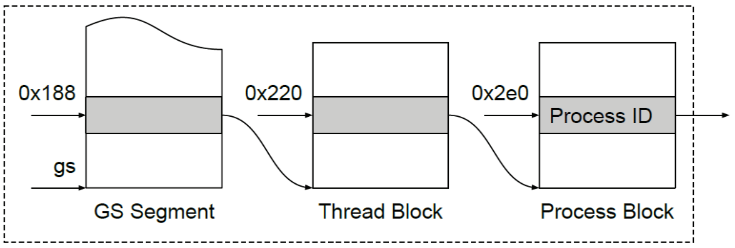


Figure 10. The GS register points to a local storage of the current processor. This local storage points to a data structures that represents the currently executing thread – the thread block. The thread block points to a data structure that represents the process which hosts the thread – the process block, which holds the identifier of the represented process.

The verification procedure can be seen as a boolean function returning true iff the verification succeeds. This function has one parameter – the virtual address that triggered the EPT violation handler. The function performs the following steps:

1. Fetch the current process identifier from OS-specific data structures. Figure 10 depicts this process on a 64-bit version of Windows 8.
2. Locate the process identifier in the memory layout data structure, which was prepared by the driver. The process descriptor contains a pointer to a list of module descriptors.
3. Locate the module descriptor that contains the virtual address that triggered the EPT violation handler. The module descriptor contains the index of the database entry that corresponds to this module.
4. Copy the module descriptor from the database to a memory region that is protected by an EPT (i.e. all types of access are restricted).
5. Validate the signature of the module descriptor.
6. Locate the information describing the page that triggered the EPT violation:
    a. Locate the section descriptor
    b. Locate the hash value of the page
    c. Locate the index of the first and the last relocations
    d. Locate the index of the first and the last datums
    e. Compute the address of the first and the last bytes described by the hash value. For example, if only the first 20 bytes of the page belong to the section, then only those bytes should be hashed.
7. Hash the page (or its part) as follows:
    a. Let p be a pointer to the first byte to be hashed
    b. Initialize pi to 0
    c. For each relocation r do:
        i. Hash the bytes [pi..r.offset-1]
        ii. Let d be the datum at offset r.offset
        iii. If d is null, fix the value at r.offset and hash it
        iv. Else, hash d.value and verify value at r.offset
        v. Advance pi to r.offset+r.length
    d. Hash the bytes [pi..the last byte to be hashed]
8. Compare the hash result to the expected hash value and return true iff they are equal

Figure 11 presents the most general example of a verification process. Datums hold the values of relocations that cross page boundaries. Since on the one hand the verification procedure must read the value at the relocation position but on the other hand it must not attempt to read data that may induce a page fault, we chose to store the values of relocation that cross page boundaries in a special array – the datums array.

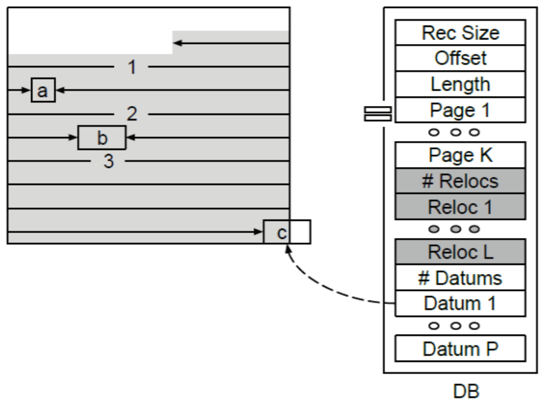


Figure 11. Page authentication in its most general form. In this case the section starts in the middle of a page. The section contains three relocations: *a*, *b* and *c*. Relocation *c* only partially belongs to the page being authenticated. The verification function first computes the hash of the bytes preceding relocation *a* (the first segment). It then subtracts from the value at position *a* the difference between the actual and the expected locations of the module and hashes the result. The same is done for relocation *b* and the second segment. Finally the verification function hashes the third segment and the relevant part relocation *c*. Since the value of relocation *c* cannot be read from the page, it is read from the datums array.

4.4 Secure execution of mixed pages

Some pages may contain both code and data. Usually, such pages appear on a boundary between a code section and a data section when those sections are not page-aligned. The problem with such pages is that on the one hand it is unsafe to grant these pages "execution" rights since they cannot be authenticated entirely, and on the other hand, the code in these pages cannot execute without "execution" rights. The solution to this problem is *controlled execution*. In essence after granting the page "execution" rights, we make sure that the control does not exit the authenticated area, by monitoring the instruction pointer. The hypervisor monitors the instruction pointer by activating the hardware debugger in a single-step mode. In this mode, the processor generates an interrupt on vector 1 after each instruction executes. The hypervisor intercepts this interrupt and checks whether the instruction pointer has left the authenticated area, and if so, the hypervisor forfeits the "execution" rights of the page.

The hardware debugger is controlled by the debug control register (DR7), the debug address registers (DR0-DR3) and the flags register. These registers define conditions in which the processor should generate a breakpoint, which is actually an interrupt on vector 1. When the defined conditions are met, the processor generates an interrupt and sets the debug status register to report the conditions that were sampled. A hypervisor can intercept interrupts and attempts to access the debug and the flags register. In other words, the hypervisor has full control of the debugging facilities and can, therefore, use these facilities securely, as will be described below.

In order to start monitoring the instruction pointer, the hypervisor sets the trap flag in the flags register and begins intercepting all interrupts (by modifying the guest IDT). After every instruction executed by the guest, a VM_EXIT occurs, enabling the hypervisor to check whether the instruction pointer is within the authenticated area. The processor clears the trap flag when an interrupt occurs, therefore the hypervisor must intercept not only the interrupt at vector 1 (the breakpoint vector) but also all the other interrupts. When an interrupt occurs, the hypervisor forfeits the "execution" rights of the partially authenticated page.

On modern processors we can improve the performance of the presented system. The IA32_DEBUGCTL MSR provides additional means to define the breakpoint conditions. Specifically when the *single-step on branches* flag (bit 1) is set (in addition to the trap flag in the flags register), the processor generates a breakpoint after every branch instruction, rather than every instruction. During instruction pointer monitoring, the hypervisor sets this flag thus intercepting all branches that may potentially transfer the control outside the authenticated area. Another way to leave the authenticated area is by falling through the last instruction. Therefore, the hypervisor installs a breakpoint on the byte following the last instruction, by writing its address to DR0 and setting the appropriate flags in the debug control register.

4. Management station

The hypervisor that was described in section 2 can prevent execution of unauthorized software by exploiting the SLAT mechanism. Obviously, the hypervisor can do so only after its activation. Therefore, the system remains vulnerable before and during its initialization: a malicious software may acquire execution rights and then either activate a malicious hypervisor or prevent activation of our hypervisor. In both cases, our hypervisor cannot provide protection against execution of such an unauthorized software. It is, therefore, desirable to inform the user about the protection status of the given system.

The management station has two responsibilities: attestation and monitoring/notification. By attestation, we mean that the management station acts as the remote key-server, attests the hypervisor that is being activated on a remote system and provides it with some secret information (i.e., cryptographic key). A detailed description of this process appears in section 3. The attestation protocol guarantees that the secret information is provided only to authentic hypervisors, which can then protect the system from unauthorized access. Therefore, possession of this secret information is a proof of the possessor's authenticity.

The second responsibility of the management station is monitoring and notification, by which we mean that the management station constantly monitors and informs the user about the protection status of remote systems, for example by displaying the statuses on the screen. The hypervisor is obligated to send a periodic message to the management station, thus indicating that the remote system is protected. The hypervisor signs its messages with the secret information that it received from the management station during the attestation protocol.

In order to prevent replay attacks, the management station generates and sends to the hypervisor a random number *s* which acts as a session id. The session id *s* is sent only once during the attestation protocol. At the *t*'s time unit the hypervisor sends to the management station a signed message containing *(s,t)*. This message proves that the hypervisor belonging to session *s* is active at time *t*. Figure 12 depicts the described protocol.

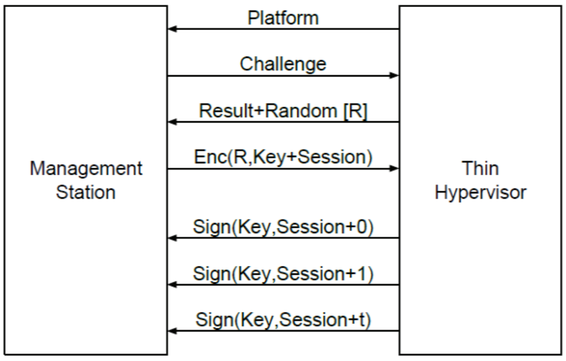


Figure 12. The protocol between the management station and the thin-hypervisor. The protocol consists of a 4-way handshake and periodic notifications. The "+" sign here means concatenation.

5. Performance

System overhead, as a result of execution protection, is attributed to actions that need to take place in the hypervisor during a VM_EXIT. This occurs when (a) execution of a write-only page is attempted and (b) as a result of a write to an execute-only page. The former's handling is more involved, since it warrants calculating the page's hash and verifying its signature, while in the latter case the operation is automatically granted. In both cases, however, the EPT needs to be updated. In single-processor environments, updating the EPT is straightforward, however, in multiprocessor environments, as previously detailed, this is more elaborate, since it requires interrupting all the other processors by activating their respective hypervisor, which in turns updates its own EPT.

The (a) and (b) intercepts, mentioned above, occur when an executable page is first executed after the application was loaded and after a page was swapped out and then back in. Therefore, overhead is also closely related to the swap activity in the system.

Performance measurements of execution-protection overhead were conducted by measuring overhead directly as well as by running well-known benchmarks on single-processor and multiprocessor systems, with and without execution protection. The benchmark suite used was the "Phoronix Test Suite" [37]. A variety of test benchmarks were selected to reflect different types of loads, such as: CPU intensive, graphics, disk-access and network.

The tests were performed on a system with the following configuration:

- Intel Core-i7-3687U@3.3GHz (4 Cores)
- 8192MB DRAM
- Intel HD4000 Graphics
- Intel 82579LM Gigabit Network
- Linux (Ubuntu 14.04 kernel 3.19.0-25 generic X86 SMP)
- GCC 4.8.4

5.1 Test A

In the first test, we measure the direct overhead associated with authorizing a writable page for execution. An executable file is mapped to memory. The executable file contains a function `void f(void)` configured on a page boundary. The first instruction in `f()` is the return instruction; The Linux `posix_fadvise()` function is called to ensure that when `f()` is called a page fault requiring a page-load from disk shall occur. This also mandates a VM_EXIT and an executable-page validation when the system is execution-protected. We measure the number of CPU cycles involved in calling `f()`. We measure 10000 calls to `f()` while execution-protection is enabled and disabled. The average number of CPU cycles required to execute `f()` without execution protection enabled was: 917021, while with execution protection enabled was: 976754. The difference, 59733 cycles, reflects the number of CPU cycles required to authenticate a page for execution.

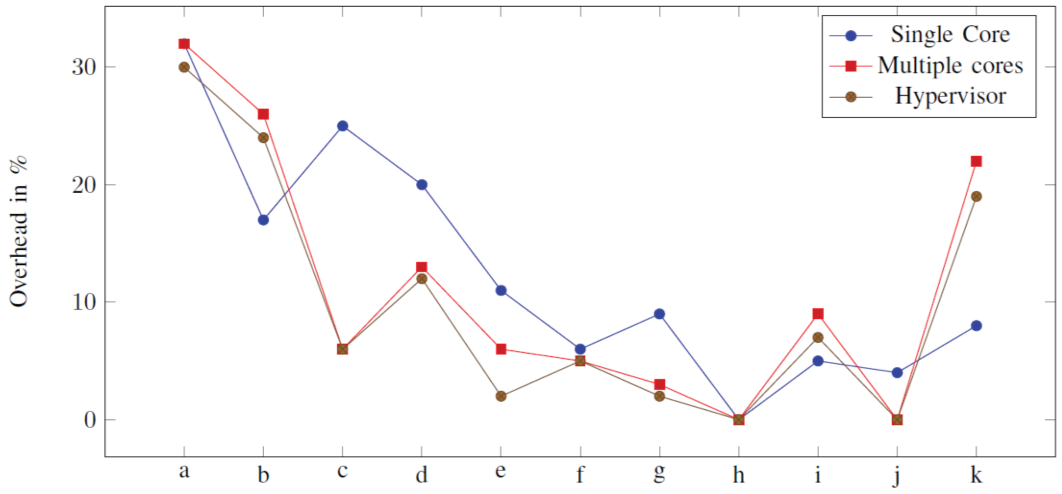


Figure 13. Overhead of the benchmark execution under different conditions: (a) single core; (b) multiple cores; and (c) with hypervisor but without execution protection

5.2 Test B

In the second test, we measure the overhead associated with executing intensive benchmarks selected from the "Phoronix Test Suite":

a) Apache – Static Web Page Serving
b) X11 – PutImage Square
c) X11 – Scrolling 500x00 px
d) X11 – Char in 80-char aa line
e) X11 – PutImage XY 500x500 Square
f) X11 – Fill 300x300 px AA Trapezoid
g) X11 – 500px Copy from Window to Window
h) X11 – Copy 500x500 Pixmap to Pixmap
i) X11 – 500Px Compositing from Pixmap to Window
j) X11 – 500px Compositing from Window to Window
k) Unpacking the Linux Kernel

To measure the effects of multiple cores, the benchmark comparisons were executed on a single core (by disabling other cores) and once again when all cores were enabled. In each case the benchmark was executed on a system with execution-protection enabled and disabled to generate the overhead comparison. The results are presented in Table 1 and depicted graphically in Figure 13.

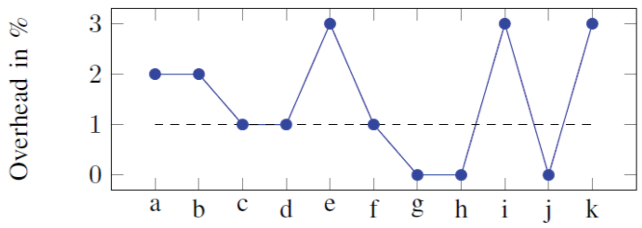


Figure 14. Overhead of execution protection only after subtraction of the hypervisor overhead. The dashed line represents the average overhead.

	Single	Multiple	Hypervisor	Net
a	32%	32%	30%	2%
b	17%	26%	24%	2%
c	25%	6%	6%	1%
d	20%	13%	12%	1%
e	11%	6%	2%	3%
f	6%	5%	5%	1%
g	9%	3%	2%	0%
h	0%	0%	0%	0%
i	5%	9%	7%	3%
j	4%	0%	0%	0%
k	8%	22%	19%	3%

Table 1. Test results

5.3 Evaluation

The results show that the total overhead of the execution-protection with a thin-hypervisor exists within a 0%-30% band, depending on the type of benchmark tested. When hypervisors are activated on systems and secondary level address translation (SLAT) is active, system overhead is caused by the additional translation required for memory access, which was measured as well. This parasitic overhead, as well as overhead caused by response to mandatory VM_EXIT events is associated with all hypervisors, however is minimized when using a thin-hypervisor. By subtracting this parasitic overhead from the general overhead values obtained for each benchmark, we present the net overhead associated with execution-protection, as can be seen in Figure 14 and in the rightmost column of Table 1. The results show an average overhead value of 1% within a 0%-3% range.

6. Conclusions

The growing threat of malicious code infiltration into computer systems is extremely grave in light of the economic losses and potential havoc they bestow. Hackers are becoming shrewder and much more cunning in their attack methodologies. They are winning the battle with the anti-malware protection industry, which is propagating an abundance of security software products geared to monitor, identify patterns and employ behavioral heuristics. As the authors point out, all Advanced-Persistency-Attacks (APTs) eventually need to execute instructions on the processor. Therefore, a suggested alternative method to eradicate most APTs is real-time monitoring and validation of executing instructions. An undertaking which can be appropriately addressed by using an attested, and therefore trusted, hypervisor. The associated total overhead is confined to 30%, where in most scenarios it is below 15%. With computer hardware performance advancing in great leaps, we believe that in return for rendering a system substantially safe from APTs, viruses, worms, buffer-overflows and malicious code injection, this overhead is justified.

References

[1] McCormack, "Five Stages of a Web Malware Attack," Sophos, Nov 2014. [Online]. Available: https://www.sophos.com/en-us/medialibrary/Gated%20Assets/white%20papers/sophos-five-stages-of-a-web-malware-attack.pdf.

[2] A. Averbuch, M. Kiperberg and N. J. Zaidenberg, "An efficient vm-based software protection," in *5th International Conference on Network and System Security (NSS)*, 2011.

[3] A. Averbuch, M. Kiperberg and N. J. Zaidenberg, "Truly-Protect: An Efficient VM-Based Software Protection," *Systems Journal, IEEE,* vol. 7, no. no. 3, p. 455–466, 2013.

[4] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Seattle, WA, USA, 2008.

[5] A. David and N. J. Zaidenberg, "Maintaining streaming video DRM," in *ICCSM*, Reading, 2014.

[6] N. J. Zaidenberg and A. David, "TrulyProtect video delivery," in *ECIW*, Jyvaskyla, 2013.

[7] R. J. Creasy, "The Origin of the VM/370 Time-sharing System," *IBM J. Res. Dev.,* vol. 25, no. no. 5, p. 483–490, 1981.

[8] C. Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," vol. 3, 2007.

[9] AMD, "AMD64 Architecture Programmer's Manual: System Programming," vol. 2.

[10] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo and K. Kato, "Bitvisor: A thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, 2009.

[11] Y. Chubachi, T. Shinagawa and K. Kato, "Hypervisor-based Prevention of Persistent Rootkits," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, 2010.

[12] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[13] M. Kiperberg, A. Resh and N. J. Zaidenberg, "Remote Attestation of Software and Execution-Environment in Modern Machines," in *CSCloud*, 2015.

[14] R. Kennell and L. H. Jamieson, "Establishing the Genuinity of Remote Computer Systems," in *Proceedings of the 12th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003.

[15] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. v. Doorn and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2005.

[16] C. Castelluccia, A. Francillon, D. Perito and C. Soriente, "On the Difficulty of Software-based Attestation of Embedded Devices," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2009.

[17] D. Schellekens, B. Wyseur and B. Preneel, "Remote Attestation on Legacy Operating Systems with Trusted Platform Modules," vol. 74, no. no. 1-2, p. 13–22, Dec 2008.

[18] A. Seshadri, M. Luk, A. Perrig, L. v. Doorn and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *Proceedings of the 5th ACM Workshop on Wireless Security*, New York, NY, USA, 2006.

[19] Y. Yang, X. Wang, S. Zhu and G. Cao, "Distributed software-based attestation for node compromise detection in sensor networks," in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, Washington, DC, USA, 2007.

[20] D. Ionescu, "Microsoft bans up to one million users from xbox live," *PC World,* 2009.

[21] Sony, "Information on banned accounts and consoles," 2015.

[22] Brian, "Nintendo starting to ban pirates from online services on 3ds," Nintendo everything, 2015.

[23] Wikipedia, "An analysis of proposed attacks against genuinity tests," [Online]. Available: http://en.wikipedia.org/wiki/Warden .

[24] D. Schellekens, B. Wyseur and B. Preneel, "Remote Attestation on Legacy Operating Systems with Trusted Platform Modules," *Sci. Comput. Program,* vol. 74, no. no. 1-2, p. 13–22, Dec 2008.

[25] S. Pearson, Trusted Computing Platforms: TCPA Technology in Context, Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[26] P. England, B. Lampson, J. Manferdelli, M. Peinado and B. Willman, "A Trusted Open Platform," *Computer,* vol. 36, no. no. 7, p. 55–62, Jul 2003.

[27] Q. Yan, J. Han, Y. Li, R. H. Deng and T. Li, "A software-based root-of-trust primitive on multicore platforms," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, New York, NY, USA, 2011.

[28] P. England, "Practical techniques for operating system attestation," in *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, Berlin, Heidelberg, 2008.

[29] E. G. a. C. J. Mitchell, "Trusted computing: Security and applications," *Cryptologia,* vol. 33, no. no. 3, p. 217–245, 2009.

[30] A. Resh and N. Zaidenberg, "Can keys be hidden inside the CPU on modern windows host," in *ECIW*, Jyvaskyla, 2013.

[31] K. K. Saluja, Linear feedback shift registers theory and applications, 1987.

[32] M. Kiperberg and N. Zaidenberg, "Efficient Remote authentication," *Journal of Information warfare,* 2013.

[33] M. Howard, M. Miller, J. Lambert and M. Thomlinson, "Windows isv software security defenses," Microsoft Corporation, 2010. [Online]. Available: https://msdn.microsoft.com/ en-us/library/bb430720.aspx.

[34] A. Dang, "Behind Pwn2Own: Exclusive Interview With Charlie Miller," March 2009. [Online]. Available: http://www.tomshardware.com/reviews/ pwn2own-mac-hack,2254-4.html.

[35] M. Pietrek, "An in-depth look into the Win32 portable executable file format," *MSDN Mag. 17, 2,* pp. 80-90, 2002.

[36] E. Youngdale, "Kernel korner: The elf object file format by dissection," *Linux Journal,* vol. 1995, no. no. 3es, p. 15, 1995.

[37] M. Larabel and M. Tippett, "Phoronix test suite," Phoronix Media, [Online]. Available: http://www.phoronix-test-suite.com/. [Accessed June 2016].

PV

SYSTEM FOR EXECUTING ENCRYPTED NATIVE PROGRAMS

by

A. Resh, M. Kiperberg, R. Leon, N.J. Zaidenberg 2017

International Journal of Digital Content Technology and its Applications (JDCTA), Vol. 11, No. 3, pp. 56-71

http://www.globalcis.org/jdcta/ppl/JDCTA3803PPL.pdf

System for Executing Encrypted Native Programs

[1]Amit Resh, [2]Michael Kiperberg, [3]Roee Leon, [4]Nezer J. Zaidenberg
[1] *Deparment of Mathematical IT, University of Jyväskylä, Finland, amitr44@gmail.com*
[2] *Faculty of Sciences, Holon Institute of Technology, Israel, mkiperberg@gmail.com*
[3] *Deparment of Mathematical IT, University of Jyväskylä, Finland, roee.leonn@gmail.com*
[4] *School of Computer Sciences, College of Management, Israel, nzaidenberg@me.com*

Abstract

An important aspect of protecting software from attack, theft of algorithms, or illegal software use, is eliminating the possibility of performing reverse engineering. One common method to deal with these issues is code obfuscation. However, in most case it was shown to be ineffective. Code encryption is a much more effective means of defying reverse engineering, but it requires managing a secret key available to none but the permissible users. The authors propose a new and innovative solution. Critical functions in protected software are encrypted using well-known encryption algorithms. Following verification by external attestation, a thin hypervisor is used as the basis of an eco-system that manages just-in-time decryption, inside the CPU, where decrypted instructions are then executed and finally discarded, while keeping the secret key and the decrypted instructions absolutely safe. The paper presents and compares two methodologies that perform just-in-time decryption: in-place and buffered execution. The former being safer, while the latter boasts better performance.

Keywords: *Hypervisor, Trusted computing, Attestation, Cyber-security*

1. Introduction

Digital content such as games, videos, and the like may be susceptible to unlicensed usage, which has a significant adverse impact on the profitability and commercial viability of such products. Commonly, such commercial digital content may be protected by a licensing verification program; these, however, may be circumvented by reverse engineering of the software instructions of the computer program which leaves them vulnerable to misuse.

One way of preventing circumvention of the software licensing program, may be using a method of obfuscation [1] [2]. The term obfuscation refers to making software instructions difficult for humans, as well as reverse-engineering software tools, to understand by deliberately cluttering the code with useless, confusing pieces of additional software syntax or instructions. However, even when changing software code and making it obfuscated, the content is still readable to the skilled hacker [3] [4].

Additionally, publishers may protect their digital content product by encryption, using a unique key to convert the software code to an unreadable format, such that only the owner of the unique key may decrypt the software code. Such protection may only be effective when the unique key is kept secured and unreachable to an adversary. Hardware based methods for keeping the unique key secured are possible [5] [6] [7], but may have significant deficiencies, mainly due to an investment required in dedicated hardware on the user side, making it costly, and, therefore, impractical. Furthermore, such hardware methods have been successfully attacked by hackers [8] [9].

Software copy-protection is currently predominantly governed by methodologies based on obfuscation, which are volatile to hacking or user malicious activities. There is, therefore, a need for a better technique for protecting sensitive software sections, such as licensing code.

In this paper, we present a system that allows encrypting and executing native programs written for the x86 architecture. The system is based on the approach proposed by Averbuch et al. [10], in which an attested kernel module is responsible for decryption and execution of encrypted functions. The main deficiency of the proposed approach is the inability of the kernel module to protect itself from the operating system. As a consequence, a vulnerability in the operating system may compromise the secret key. Moreover, the attestation server has to attest not only the kernel module responsible for decryption but also the entire operating system. The complications of operating system attestation and a partial mitigation are described in [11].

This paper proposes to solve all these complications by utilizing the virtualization extension, which is available on modern processors [12] [13], in order to enable the decrypting kernel module to protect itself, thus eliminating the need for operating system attestation. Figure 1 depicts the components of the proposed system as well as their relationships. The system is deployed on three computers: a development machine, on which the program to be encrypted, is compiled and encrypted; the attestation server, which stores the decryption key, and delivers it to the target machine; and the target machine, which executes the encrypted program. A special driver, which embeds a hypervisor, is installed on the target machine prior to execution of an encrypted program. The hypervisor obtains the decryption key, which is necessary for program execution, from the attestation server, when an encrypted program is loaded to the memory.

1.1 Intel SGX

Intel has announced its new security technology named Software Guard Extensions (SGX) [32], which enables developers to create secure containers, called enclaves, inside a process address space. The enclave address space is protected from any other software not resident in the enclave, including privileged software. This guarantees that malware, at any privilege level, cannot compromise the confidentiality or integrity of enclave resident software or data. SGX does not rely on a hypervisor or hardware virtualization, instead it encompasses two new instruction-set extensions that allow initializing and managing the enclaves. Secure storage is managed in an Enclave-Page-Cache, which is protected by hardware from "non-enclave" access. SGX provides the means for implementations to the same end as proposed by our methodology, however the SGX processor extensions are available only in the newest Intel processors. Therefore, utilizing an SGX based solution requires specific hardware, adds to equipment cost and is not supported on legacy systems.

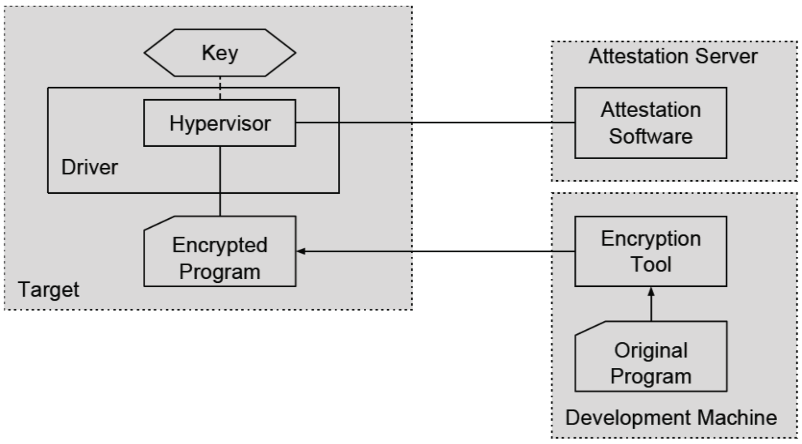


Figure 1. Native code protection system. The original program is encrypted before its distribution. The encryption key is stored in the attestation server, which delivers it to the hypervisor in the target machine upon successful attestation. The hypervisor is initialized by a driver, which also hosts the code of the hypervisor.

1.2 Contribution

The methodology proposed in this paper provides for a software-only solution, based on the availability of hardware virtualization and secondary-level address translation, incorporated in most Intel and AMD CPUs released after 2008. Furthermore, an innovative thin hypervisor is utilized to protect cryptographic keys and decrypted code to provide a truly secure just-in-time code decryption mechanism. The thin hypervisor is guaranteed to be trusted with the employment of remote attestation.

2. Encryption tool

The encryption tool is responsible for encryption of selected functions in a program. The user selects the functions to be encrypted by specifying their names in a configuration file. A *map file* or a *debug symbols file*, which are produced by a compiler, can then be used to translate the names of the functions to their locations in the program file.

On Windows, program files, executables and dynamic libraries, are stored in Portable Executable (PE) format [14]. Figure 2 depicts the structure of a PE file. The different headers define the expected location of the PE file when loaded to memory, sizes and positions of various data structures inside the PE file, the number of sections contained in this PE file, etc. The section table contains a description of each of the sections contained in the PE file. Following the section table are the sections themselves. Sections vary in their structure and purpose: the .text section contains the code of the program, the .data section contains its constants. Other sections may contain information about resources (images and sounds) embedded in the PE file or information used during exception delivery.

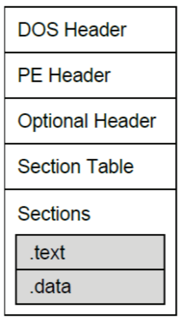


Figure 2. Structure of a Windows PE file. The structure contains a variable number of sections. Two of the most common sections are presented.

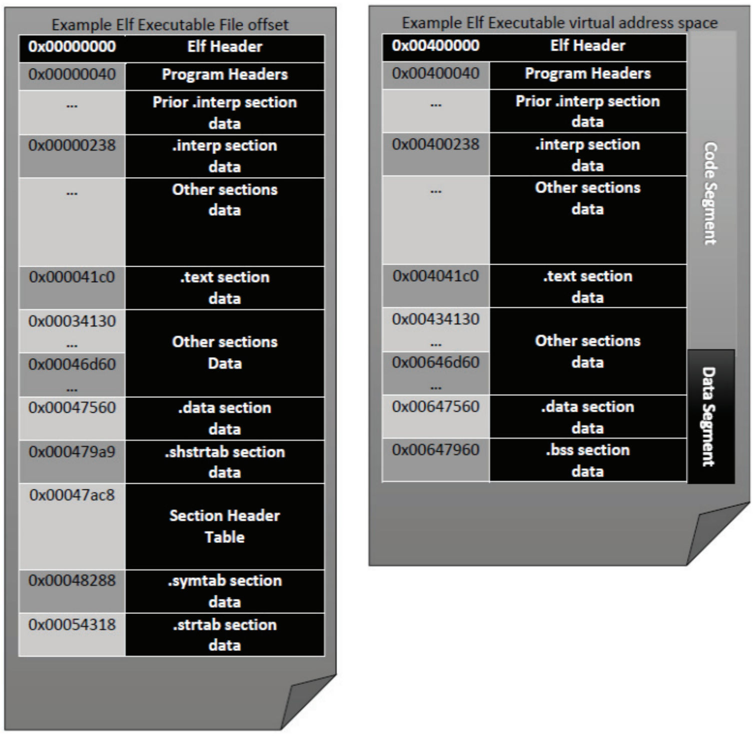


Figure 3. The left image represents the structure of an ELF file as it is stored in disk. The right image represents the structure of an ELF file as it is loaded to memory.

On Linux, program files, executable files and dynamic libraries, are stored in Executable and Linkable Format (ELF) format [15]. Figure 3 depicts the structure of an ELF file. An ELF file consists of a header, which is followed by data. The data may include:

- Program header table, describing zero or more segments. Only two segments can be defined as loadable: the code segment and the data segment. The code segment is loaded to memory with read-write-execute permissions, while the data segment is loaded with read-only permissions. Other segments are not loaded to memory.
- Section header table, describing zero or more sections. A typical ELF file holds a section called *.text*, which contains the code of the program.
- Data referenced by entries in the program header table or section header table.

The segments contain information that is necessary for runtime execution of the file, while the sections contain data for linking and relocation. Figure 3 depicts the structure of an ELF virtual-image at load time.

The encryption tool modifies the given PE/ELF file by introducing a new section, which stores the selected functions in encrypted form. The instructions of the original functions are partially replaced by an exception inducing instruction. We propose to use either the *halt* instruction or the *software breakpoint* instruction. The halt instruction is a privileged instruction, which deactivates the current processor when executed in kernel mode, but generates a general protection fault when executed in user mode. The software breakpoint instruction generates a breakpoint trap when executed in either kernel or user modes. Faults and traps, being types of interrupts, can be intercepted by a hypervisor, which can then decrypt and execute the original encrypted function. Another benefit of the halt and the software breakpoint instructions is that they can be represented by a single byte (0xF4 for halt and 0xCC for software breakpoint), thus allowing them to fully cover any number of bytes. The software breakpoint instruction is superior to the halt instruction in that it generates an interrupt not only in user mode but also in kernel mode.

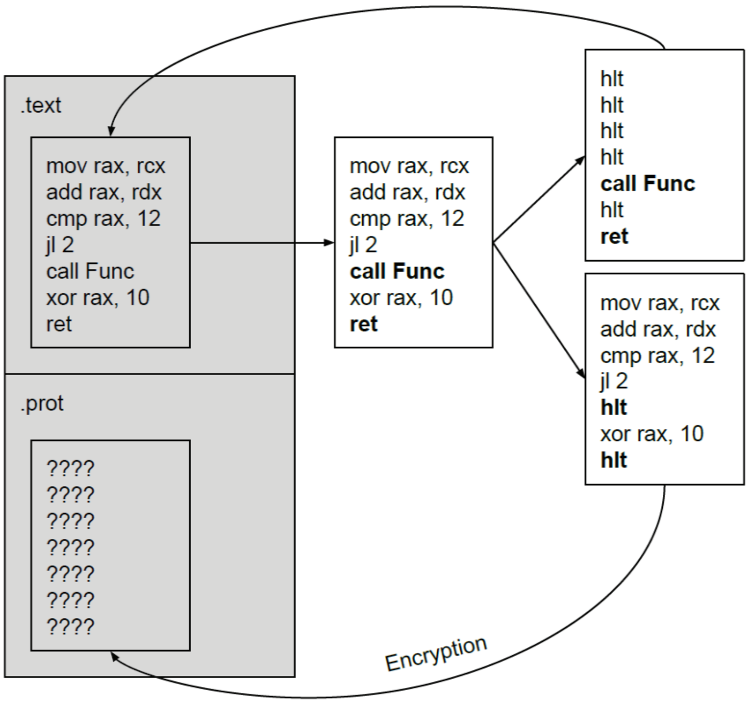


Figure 4. Example of an encryption process of a single function. The encryption begins by classifying instruction is encryptable (normal face) and non-encryptable (bold face), and creating to copies. The complementary instructions in each copy are replaced by halts. Finally, one copy is written over the original functions, and the other is encrypted and added to the special section.

As will be explained in section 5, it is highly important to intercept control transfers that leave the encrypted function. The encryption tool disassembles the function to be encrypted and inspects its instructions. The instructions then are classified as *encryptable* and *non-encryptable*. The encryption tool classifies an instruction as non-encryptable if it might transfer control out of the encrypted function. For example, the ret and the call instructions are always classified as non-encryptable, but the jmp instruction is classified as non-encryptable only if its destination lays outside of the protected function's bounds or if the destination cannot be determined statically (if it is stored in a register, for instance).

The encryption tool produces two copies of the original function, the encryptable copy (EC) and the non-encryptable copy (NEC). In the EC all the non-encryptable instructions are replaced by the halt or the software breakpoint instructions. Then the encryption tool encrypts the EC and stores it in the new section. In the NEC all the encryptable instructions are replaced by the halt or the software breakpoint instructions. Then the encryption tool replaces the original function by the NEC. Figure 4 presents an example of such a transformation.

3. Hypervisor

A hypervisor, also referred to as a Virtual Machine Monitor (VMM), is software, which may be hardware-assisted, to manage multiple virtual machines on a single system [16]. The hypervisor virtualizes the hardware environment in a way that allows several virtual machines, running under its supervision, to operate in parallel over the same physical hardware platform, without obstructing or impeding each other. Each virtual machine has the illusion that it is running unaccompanied on the entire hardware platform. The hypervisor is referred to as the *host*, while the virtual machines are referred to as *guests*.

A virtual machine control structure (VMCS) is defined for each virtual environment managed by a virtual machine monitor (VMM) [12]. This structure defines the values of privileged registers, the location of the interrupt descriptors table, and additional values that constitute the internal state of the virtual environment. In addition, this structure defines the events that the VMM is configured to intercept, and the address of the function that should handle the interception. The act of control transfer from the virtual environment to a predefined function is called vm-exit and the act of control transfer from the function back to the virtual environment is called vm-entry. Upon vm-exit the function can determine the reason of the vm-exit by examining the fields of the VMCS and altering them, thus altering the state of the virtual environment as it wishes. Finally, the VMCS can define a mapping between the physical memory as it is perceived by the virtual environment and the actual physical memory. As a consequence, the VMM can prevent access to some physical pages by the virtual environment. Moreover, the virtual environment will be unaware of this situation.

We propose to use a hypervisor for securing a single guest. Rather than wholly virtualizing the hardware platform, a special breed of hypervisor, called a *thin hypervisor*, is used [17] [18]. A thin hypervisor is configured to intercept only a small portion of events. All other events are processed without interception, directly, by the OS. A thin hypervisor only intercepts the set of events that allows it to protect an internal secret (such as a cryptographic key) and protect itself from subversion. Figure 5 depicts a thin hypervisor supporting a single guest. Since a thin hypervisor does not control most of the OS interaction with the hardware, multiple OS are not supported. On the other hand, system performance is kept at an optimum.

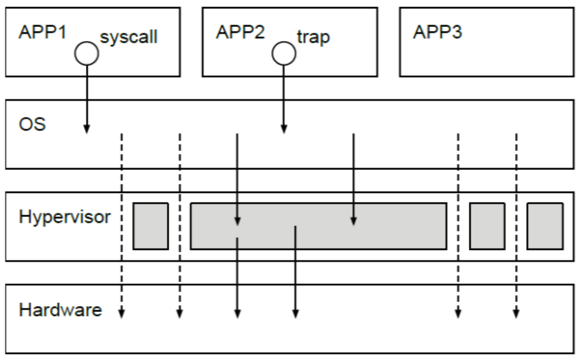


Figure 5. Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions, and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service from the underlying hardware. A thin hypervisor can intercept some of those requests and handle them according to some policy.

A thin hypervisor facilitates a secure environment by: (a) setting aside portions of memory that cannot be accessed by the guest, (b) storing the cryptographic key in privileged registers, and (c) intercepting privileged instructions that may compromise its protected memory, reveal the cryptographic key, or attempt to subvert the hypervisor.

Once this environment is correctly configured, a thin hypervisor can be utilized to carry out specific operations, which may include use of the cryptographic key, in a protected region of memory. As a result of the tightly configured intercepts and absolute control of the protected memory regions, this activity can be guaranteed to protect both the cryptographic key and the operations results.

4. Remote attestation

The problem of remote software authentication, determining whether a remote computer system is running the correct version of a software, is well known [5] [19-25][33]. Equipped with a remote authentication method, a service provider can prevent an unauthenticated remote software from obtaining some secret information or some privileged service. For example, only authenticated gaming consoles can be allowed to connect to the gaming networks [26-28], and only authenticated bank terminals can be allowed to fetch records from the bank database [29]. We have also shown that once attestation is completed the attested computer can receive encryption keys from the attestation server and protect them from malicious software in a modern host [34].

The research in this area can be divided into two major branches: hardware assisted authentication [5-7] and software-only authentication [19-22]. In this paper we concentrate on software-only authentication, although the system can be adapted to other authentication methods, as well. The authentication entails simultaneously authenticating some software component(s) or memory region, as well as verifying that the remote machine is not running in virtual or emulation mode. Software-only authentication methods may also involve a challenge code that is sent by the authentication authority, and executed on the remote system. The challenge code computes a result that is then transmitted back to the authority. The authority deems the entity to be authenticated if the result is correct and was received within a predefined time-frame. The underlying assumption, which is shared by all such authentication methods, is that only an authentic system can compute the correct result within the predefined time-frame. The methods differ in the means by which (and if) they satisfy this underlying assumption.

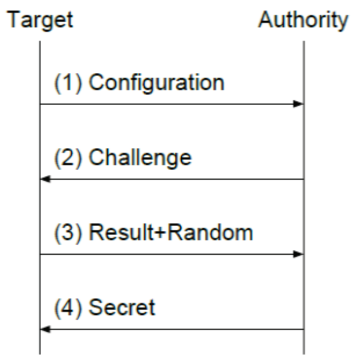


Figure 6. The attestation protocol between the authentication authority and the target machine. The protocol consists of four messages. The first two messages are sent unencrypted, while the two last messages are encrypted. The third message is encrypted by the public key of the authentication authority and the fourth message is encrypted by the random value transmitted in the third message.

Kennell and Jamieson proposed [19] a method that produces the result by computing a cryptographic hash of a specified memory region. Any computation on a complex instruction set architecture (Pentium in this case) produces side effects. These side effects are incorporated into the result after each iteration of the hashing function. Therefore, an adversary, trying to compute the correct result on a non-authentic system, would be forced to build a complete emulator for the instruction set architecture to compute the correct side effects of every instruction. Since such an emulator performs tens and hundreds of native instructions for every simulated instruction, Kennell and Jamieson conclude that it will not be able to compute the correct result within the predefined time-frame. The method of Kennel and Jamieson was further adapted, by the authors, to modern processors [30]. The adaptation solves the security issues that arise from the availability of virtualization extensions and multiplicity of execution units.

The authentication protocol is depicted in Figure 6. The initial messages of the protocol carry information about the current configuration of the target machine. Following this exchange, the authentication authority transmits a message containing the challenge code to be executed on the target machine. The target machine executes the challenge, which computes a result that is a cryptographic hash of some memory region, possibly with some additional information. The target machine, concatenates a randomly generated number to the result, encrypts both values with the public key of the authentication authority, and transmits the encrypted message. The authentication authority verifies that the result is correct and was received within a predefined time-frame. If both are true the target machine is considered authentic. The authentication authority then shares some secret information with the target machine. This secret information constitutes a proof of the target's authenticity. The authentication authority encrypts the secret information with a random value obtained from message (3) used as the encryption key, and transmits the encrypted message to the target machine.

5. Encrypted instructions execution

In order to execute an encrypted program, the user must first install the driver, which encapsulates the hypervisor. The driver monitors the PE files (ELF files, in Linux) loaded by the OS, and keeps track of PE files that contain the special encrypted functions section. When the first such PE file is loaded, the driver initializes the hypervisor. During the initialization, the driver communicates with the authentication authority, passes the attestation verification, obtains the cryptographic key, and enters a virtualized state.

The hypervisor is configured to intercept the general protection fault. When a protected program transfers control to an encrypted function, the processor attempts to execute the halt instruction, which induces a general protection fault, thus transferring control to the hypervisor. General protection faults rarely occur during the normal course of program execution, since they usually cause the program to terminate abruptly. Nevertheless, the hypervisor uses the data structures prepared by the encryption tool to test whether the general protection fault occurred during execution of an encrypted function.

The hypervisor injects the interrupt back to the guest, if it was not caused by an encrypted function execution. Otherwise, the hypervisor decrypts the function and starts its execution. Since during its execution, the function is stored in memory in unencrypted form, it is highly important to ensure that no other code has access to the decrypted instructions of the function. We note that in modern processors, several execution units (logical processors) can execute programs concurrently. Therefore, we must ensure that programs executed by all execution units have no access to the unencrypted instructions.

We present two approaches to sensitive functions execution: *in-place execution* and *buffered execution*.

5.1 In-place execution

According to this approach the hypervisor can be in one of two states: ***cold*** or ***hot***. In the cold state the memory does not contain any sensitive information and only the cryptographic key and the hypervisor's state must be protected. This is the regular mode of operation described in section 3. The hypervisor switches to the hot state when the memory contains sensitive information, which cannot be protected by the normal hypervisor memory protection technique (for example, based on EPT), since its physical location is not known (or not constant). EPT (Extended Page Table) is a secondary address translation facility used by the hypervisor to translate guest physical addresses to actual physical addresses. Switching to hot mode occurs when the hypervisor triggers execution of a decrypted function.

In the following description, we assume that the encryption tool uses ***halt*** as a replacement opcode, but the same is true when the ***software breakpoint*** opcode is used.

At initialization the hypervisor's state is set to cold. In this state, in addition to the regular protection means described in section 3, the hypervisor intercepts general protection faults. An encrypted function, which was overwritten by the NEC consists mainly of halt instructions. Execution of any of these instructions induces a general protection fault, which causes a vm-exit and transfers control to the hypervisor. The hypervisor inspects the source of the general protection fault, and fetches the EC that corresponds to this NEC. Then the hypervisor switches to hot mode and decrypts the EC into its natural location, currently occupied by the NEC (the NEC is saved in a different location for future use).

During the switch to hot mode, the hypervisor freezes all other execution units, and configures itself to intercept all interrupts. This behavior guarantees that the function in its decrypted form cannot be read by any other, potentially malicious, code, simply because no other code can run in hot mode. We note that all the control transfer instructions in the EC are replaced by the halt instruction, which induces a vm-exit.

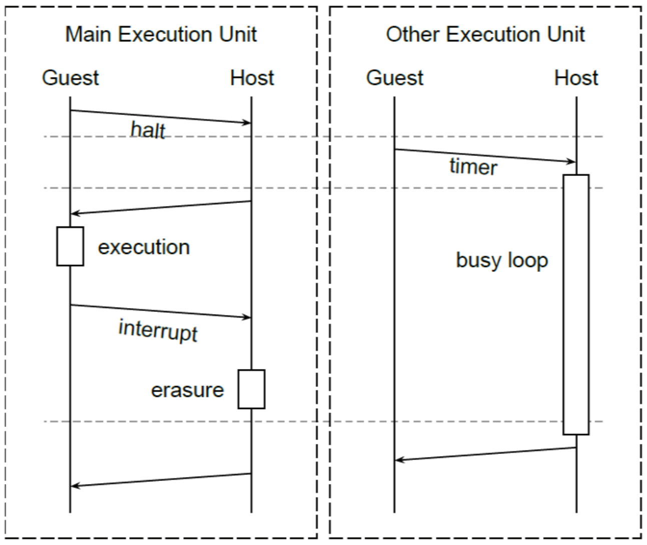


Figure 7. Example of encrypted function execution. The figure depicts two execution units, each with two alternating states: guest and host. The dashed horizontal lines are synchronization barriers, i.e. everything above the line is guaranteed to complete before anything below the line starts.

When a vm-exit occurs in hot mode, the hypervisor first replaces the decrypted function with the NEC, and switches to cold mode. Following this, the hypervisor resumes all the execution units, configures itself to intercept only general protection faults, and returns control to the guest. Figure 7 depicts the control flow during encrypted function execution.

We suggest to freeze other execution units by inducing a vm-exit on each execution unit, and running a busy loop until the hypervisor switches back to cold mode. A vm-exit can be induced either implicitly with a timer or explicitly by sending an inter-processor interrupt (IPI). The former solution is much easier to implement but the later solution is much more efficient.

The hypervisor intercepts interrupts in hot mode by replacing the original interrupt descriptor table (IDT) of the OS with a specially crafted IDT. In this special IDT each handler induces a vm-exit, for example, by executing the CPUID instruction. The hypervisor intercepts this instruction, realizes that an interrupt at vector *N* occurred and switches to cold mode. The hypervisor proceeds by installing the original IDT and moves the guest's instruction pointer to point to the N^{th} interrupt handler of the original IDT.

5.2 Buffered execution

In the following description, we assume that the encryption tool uses halt as a replacement instruction for NECs and software breakpoint as a replacement instruction for ECs.

According to this approach, the hypervisor has only one state, in which it protects itself as described in section 3. In addition, the hypervisor configures itself to intercept general protection faults. Execution of halt instructions induces a general protection fault, which causes a vm-exit and transfers control to the hypervisor. The hypervisor inspects the source of the general protection fault, and fetches the EC that corresponds to this NEC.

When the EC is resolved, the hypervisor decrypts it into a pre-allocated memory buffer, which is protected by the hypervisor's second-level translation tables (EPT). The decrypted EC will be executed in host mode, thus allowing it to reside in an EPT-protected buffer. Since the decrypted instructions are inaccessible by any other execution unit (in guest mode), there is no need to suspend them. Likewise, since the encrypted instructions are executed inside the hypervisor, there is no need to modify the IDT of the guest. Finally, there is no need to perform the costly transitions to and from the guest after every decryption. All these improve the overall performance of the system by a large factor.

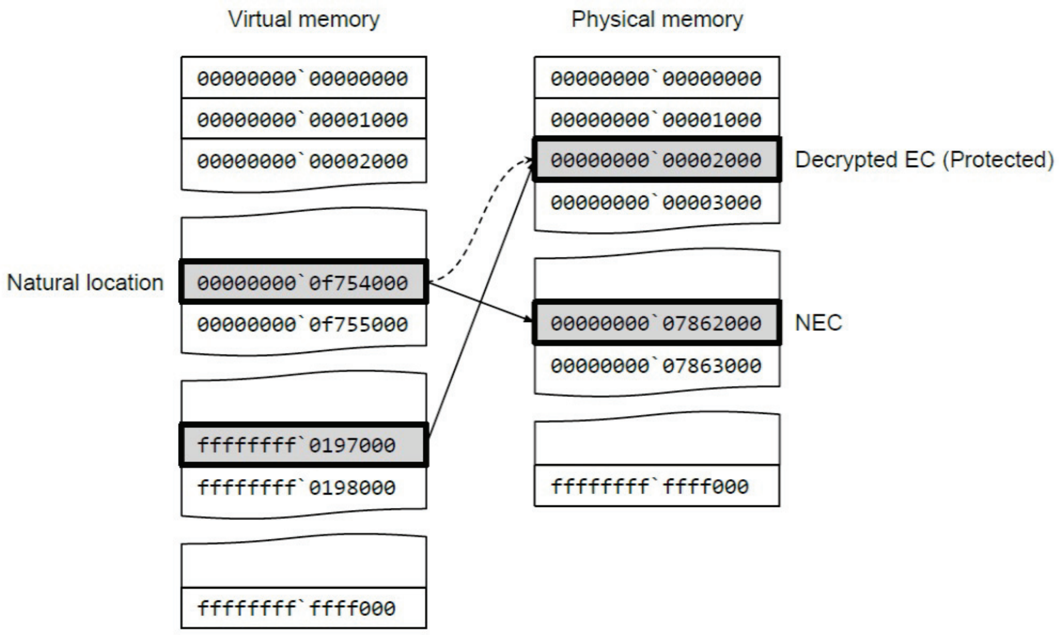


Figure 8. Memory layout during buffered execution. The functions resided at virtual address f754000, which is mapped to the physical address 7862000. The encrypted code is decrypted to virtual address ffffffff`0197000 which is mapped to the physical address 2000. The hypervisor changes the mapping of the virtual address f754000 to map the physical address 2000.

The x86 instruction set architecture defines many memory access instructions as *relative*, meaning that their arguments should not be interpreted as actual memory locations but rather they should be interpreted as offsets from the current value of the instruction pointer. As a consequence, the same instruction may have different interpretations when executed at different locations. Therefore we must execute the decrypted EC at its natural location. In order to achieve this, the hypervisor modifies the virtual page table of the current process by mapping the virtual page containing the NEC to the physical address of the pre-allocated buffer containing the decrypted EC. Figure 8 depicts this transformation.

The control flow during the execution of an encrypted function is illustrated in Figure 9. The process begins when an encrypted function is called. The first instruction in the NEC is the halt instruction; its execution triggers the general protection exception, which induces a vm-exit. The hypervisor prepares the system for buffered execution by performing the following steps: (1) the EC is decrypted into a pre-allocated buffer; (2) the virtual page table is modified to map the natural location of the function to the pre-allocated buffer, as illustrated in Figure 8; (3) the values of the guest registers, which were stored during the vm-exit transition, are restored; (4) the decrypted function is called. The decrypted function executes until an interrupt occurs. The interrupt can be triggered by a software breakpoint instruction or by some other condition, e.g., a page fault. In both cases the hypervisor suspends the buffered execution by performing the following steps: (1) the values of the registers are stored to a memory region from which they will be restored during vm-entry; (2) the virtual page table is restored to its original state; (3) the decrypted EC is erased. If the interrupt was triggered by a software breakpoint instruction, the hypervisor resumes the guest immediately. However, if the interrupt was triggered by some other condition, the hypervisor injects the interrupt to the guest, and then resumes it. The interrupt injection mechanism allows the hypervisor to delegate the responsibility of interrupt handling to the operating system. Figure 9 illustrates the simple case of software breakpoint interrupt.

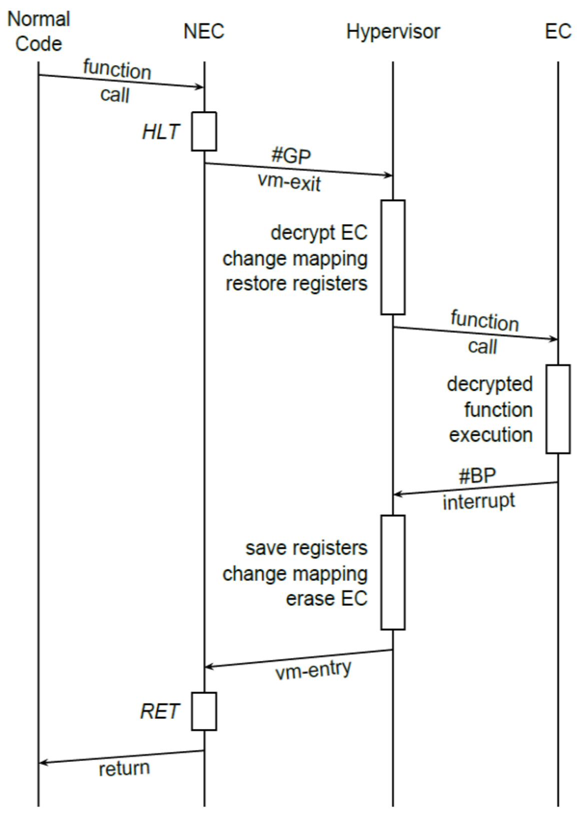

Normal Code
NEC
Hypervisor
EC
function call
HLT
#GP vm-exit
decrypt EC change mapping restore registers
function call
decrypted function execution
#BP interrupt
save registers change mapping erase EC
vm-entry
RET
return

Figure 9. Example of encrypted function execution in buffered execution mode. The figure depicts the control flow during the execution of an encrypted function.

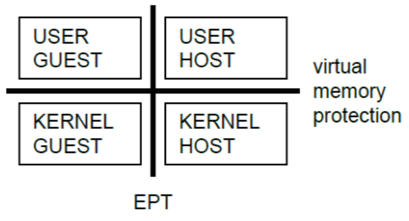


Figure 10. Execution modes. The left column represents the guest mode, while the right column represents the host mode. The lower row represents the kernel mode, while the upper row represents the user mode. The host mode can protect itself from the guest mode through the EPT mechanism. The kernel mode can protect itself from the user mode through the virtual memory protection mechanism.

This approach is more efficient but potentially less secure than the in-place execution. According to this approach, the decrypted functions are executed inside the hypervisor itself. As a consequence these functions have the same privileges as the hypervisor. In particular, they can read and write memory, which is otherwise inaccessible to any code external to the hypervisor. One can argue that it is impossible for an adversary to replace the EC with random code, without knowing the cryptographic key. However unfortunately, it is possible that some memory manipulation can be performed indirectly by modifying the data on which the encrypted function works. Nevertheless, although possible, it seems to be extremely difficult to manipulate the behavior of unknown code through its data. Possible solutions to this problem will be discussed in our future research.

6. Performance

This section presents a performance analysis of the two execution methods that were described in section 5.

We first measured the direct overhead associated with executing an encrypted function. To do that we created a function `f()` of size 128 bytes. The function's first instruction is a return instruction, therefore, once activated, the function immediately returns to the caller. In the executable file we encrypt `f()` and measure the number of CPU cycles used in a call to `f()`. Since `f()` is encrypted, calling `f()` entails a transfer from "cold" mode to "hot" mode, i.e. VM_EXIT to the hypervisor, decryption of `f()`'s contents execution of `f()` (in this case basically zero cycles since the first instruction is an immediate return) and then restoring to "cold" mode. Measurements of this full-cycle were averaged over 10000 trials with an average of 7100 cycles when using "buffered" mode and 23,000 cycles when using "in-place" mode.

To measure the overhead associated with real-world applications, we decided to use standard benchmarks as the model. The measurements were performed by encrypting several of the major functions in standard benchmark programs and comparing the performance results of each benchmark when executed with and without those functions encrypted. Two performance measurements were obtained for benchmarks that were run with an encrypted function: (a) using "In-Place Execution" and (b) using "Buffered-Execution".

System overhead, as a result of running encrypted code over the hypervisor, is attributed to actions that need to take place in the hypervisor during a VM_EXIT. This occurs when (a) an encrypted function is called; (b) a call is made from within an encrypted function to a non-encrypted function; a return occurs from the calls in (a) or (b). In (a) the function needs to be decrypted and the processor is put into "hot" mode: when the "In-Place" method is used other processors need to be frozen; when "buffered" mode is used the hypervisor needs to remap the execution pages. In (b) and (c) the operation is reversed by clearing decrypted-memory and putting the processor back into "cold" mode. Therefore, overhead is closely related to the number of transitions into and out of "hot" mode.

Additional overhead can be observed as a result of activating the hypervisor without regard to activities required to support executing encrypted software. This overhead is attributed to the fact that the system is running over a hypervisor, which activates *secondary level address translation* (SLAT) that implies overhead as a result of the additional translation required for memory access, as well as needing to intercept some mandatory events.

Performance measurements of encrypted software execution overhead were conducted by running well-known benchmarks on a multiprocessor system with and without encrypted functions.

We chose the "Phoronix Test Suite" [31] as our benchmark suite. A variety of test benchmarks were selected to reflect different types of loads, such as: CPU intensive, graphics, disk-access and network activities. The tests were performed on a system with the following configuration:

- Intel Core-i7-3687U@3.3GHz (4 Cores)
- 8192MB DRAM
- Intel HD4000 Graphics
- Intel 82579LM Gigabit Network
- Linux (Ubuntu 14.04 kernel 3.19.0-25 generic X86 SMP)
- GCC 4.8.4

We have performed three tests. In each test, we have selected an application and encrypted several central functions. Table 1 summarizes the information about the encrypted function in each application.

The first application, "Parallel BZIP2 Compression", is CPU intensive. It measures the time needed to compress a file (a .tar package of the Linux kernel source code) using BZIP2 compression. The second application, "Unpacking the Linux Kernel", measures how long it takes to extract the .tar.bz2 Linux kernel package. The third application is "X11 – 500px PutImage Square". The package "x11perf" is a very basic performance/regression test for X.Org (Window System).

Each of the benchmark tests was executed after a full system reboot (to ensure a "clean" system) and measured under the following conditions: (a) non-encrypted executable without a hypervisor active; (b) non-encrypted executable with a commercial hypervisor (VMWare) active; (c) non-encrypted executable with TrulyProtect thin-hypervisor active; (d) Encrypted executable using "In-Place" mode; and (e) Encrypted executable using "Buffered" mode. Each activation of a "Phoronix Test Suite" benchmark generates multiple runs of the benchmark to gather significant statistics.

Table 2 presents the results that were measured during benchmark execution in various configurations. The two leftmost columns describe the configuration in which the test was executed. The third column specifies the parameter that was measured. The three rightmost columns contain the values that were measured for each parameter. The table is divided into five parts: (a) No hypervisor – where measurements were performed on a non-encrypted executable without an active hypervisor; (b) vmWare HV active and KVM HV active – where measurements were performed on a non-encrypted executable with a commercial hypervisor (vmWare and KVM); (c) TP HV Active – where measurement were performed with TrulyProtect thin-hypervisor; (d) Overhead Calculation – this part summarizes the first three parts; (e) Net overhead calculations – this part presents the overhead of the in-place and the buffer decryption methods after subtraction of the overhead associated with TrulyProtect hypervisor.

Application	Function name	Size (in bytes)
Parallel BZIP2 Compression	BZ2_bzBuffToBuffCompress	317
	BZ2_bzCompressInit	588
	BZ2_bzCompress	380
	BZ2_bzCompressEnd	123
Unpacking the Linux Kernel	extr_init	94
	run_decompress_program	443
	tar_checksum	175
	extract_finish	47
	checkpoint_finish	63
X11 500px PutImage Square	InitPutImage	93
	InitGetImage	140
	DoPutImage	350

Table 1. Encrypted functions summary.

The third part is further subdivided into three parts: (i) Non protected – where a non-encrypted executable was measured; (ii) In-Place – where an encrypted executable was executed using the in-place decryption method; (iii) Buffered – where an encrypted executable was executed using the buffered decryption method.

The fourth part compares the execution times of a non-encrypted executable to four other modes of execution: (i) a non-encrypted executable while a commercial hypervisor is active; (ii) a non-encrypted executable while TrulyProtect thin-hypervisor is active; (iii) an encrypted executable which is executed using the in-place decryption method; (iv) an encrypted executable which is executed using the buffered decryption method. A graphical representation of this data appears in figures 11. Figure 12 presents the overhead of the in-place and the buffer decryption methods after subtraction of the overhead associated with TrulyProtect hypervisor.

Overhead was calculated by solving for the degradation in percent relative to the reference benchmark result as measured without the hypervisor activated.

			Parallel BZIP2 Compression	Unpacking the Linux Kernel	X11 500px PutImage Square
No HV	Not Protected	Execution	26.58 secs	10.31 secs	2822 ops/sec
vmWare HV Active	Not Protected	Execution	28.92 secs	14.83 secs	1643 ops/sec
KVM HV Active	Not Protected	Execution	28.39 secs	11.4 secs	905 ops/sec
TP HV Active	Not Protected	Execution	26.92 secs	11.81 secs	2795 ops/sec
	In-Place	Execution	31.74 secs	16.6 secs	1997 ops/sec
		VM_EXITs	222	129663	170857
		Decryptions	64	64743	85263
	Buffered	Execution	27.07 secs	12.05 secs	2667 ops/sec
		VM_EXITs	174	64743	107316
		Decryptions	64	64743	107316
Overhead Calculations	vmWare HV		9%	44%	42%
	TP HV		1%	15%	1%
	In-Place		19%	61%	29%
	Buffered		2%	17%	5%
Net Overhead	In-Place		18%	46%	28%
	Buffered		1%	2%	5%

Table 2. Test results.

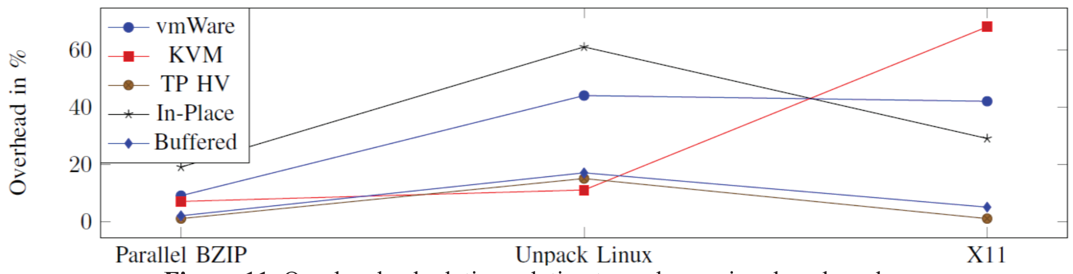


Figure 11. Overhead calculation relative to no-hypervisor benchmarks.

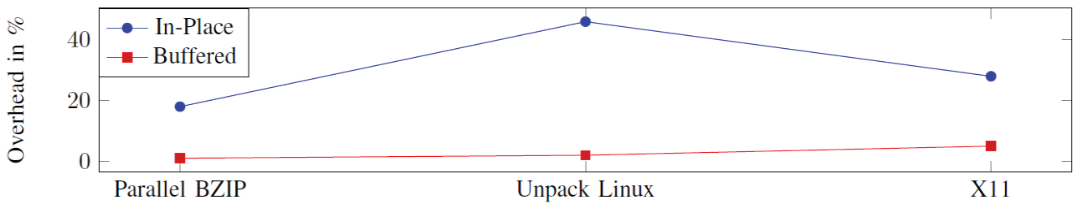


Figure 12. Net encrypted execution overhead.

7. Future work

As was explained above, the buffered execution method is superior to the in-place execution method in terms of performance. Unfortunately, the buffered execution method allows an adversary to access regions of memory that are normally protected by the hypervisor. Consider the *memcpy* function, for example. Assume that this function is encrypted and is now being executed by the hypervisor in buffered execution mode. By specifying the address of the VMCS structure in the *source* or *destination* argument, an adversary can inspect and modify the control structures of the hypervisor. Moreover, since the hypervisor executes in kernel mode, the protected function can access OS memory region and execute privileged instructions.

Fortunately, the x86 instruction set architecture provides a great variety of memory protection mechanisms, which can be utilized by the buffered execution method. One such mechanism is the virtual memory protection, which is available in both 32- and 64-bit execution modes. The virtual memory protected mechanism allows to specify a separate set of accessibility rights for kernel mode and user mode. Similarly, the hypervisor's memory protection (virtualization, to be precise) mechanism, called the Extended Page Table (EPT) on Intel processors, allows to specify a separate set of accessibility rights for host mode and guest mode. The different modes of execution and the protection mechanisms are summarized in Figure 10.

The in-place execution method utilizes the EPT to protect hypervisor's control structures and other sensitive data from an adversary. We propose to use the virtual memory protection mechanism in the buffered execution method. In particular, the buffered execution method can execute the decrypted function in user mode inside the host mode (the upper right block in Figure 10); this mode is not used by the system described in this paper. In this mode we can prevent attempts to execute privileged instructions or access the hypervisor's control structures.

The hypervisor can transit to this mode by executing the iret instruction, which is usually used to terminate an interrupt handler. This instruction modifies the execution location and the execution mode (from kernel to user). Since the execution takes place in host mode, interrupts cannot be intercepted by the hypervisor through configuration of the VMCS. The hypervisor is forced to use the IDT, which allows the kernel to specify the interrupt service routines for each of the 256 interrupt vectors. Upon interrupt, the interrupt service routine can decide whether to handle the interrupt inside the hypervisor or inject it to the guest.

We believe that the described approach will substantially improve the security of the buffered execution method, thus making it absolutely superior to in-place execution.

8. Conclusions

We present research pertaining to the methodologies of executing encrypted native machine-code, where decryption and execution are done on the fly and secure with a thin hypervisor. Two alternative methods are considered: *in-place* and *buffered* – that trade security for performance. The in-pace method executes decrypted-code in guest mode, thereby limiting the functionality of the decrypted function to whatever a guest may perform. In buffered execution method, the decrypted function executes in host mode, potentially incurring the risk of a rogue implementation accessing sensitive memory areas. For this reason the in-place method is considered safer. However, in modern multi-processor systems, the in-place method requires controlling (freezing) other execution units, while a single execution unit executes decrypted code. This requires larger overhead when compared to the buffered method and thus has a performance toll. Larger overhead is expected to be more significant for larger functions. The reason for this is related to the fact that overhead is acquired during transitions between cold to hot and hot to cold modes in the in-place method, as compared to transitions between host-execution of decrypted code and guest-execution of interrupts. Larger functions acquire more transitions, therefore overhead is more prominent in the in-place method. Given these results our conclusions are to use the (safer) in-place methodology for short functions (smaller than 1000 bytes). For larger functions (larger than 1000 bytes), allow a user-defined switch in the encryption tool to prefer security, in which case in-place shall be used, or performance, in which case buffered shall be used. In future work we plan to augment the buffered method to overcome its potential security flaws and render it the single and best alternative to use.

9. References

[1] Themida, http://www.oreans.com/, Oreans.
[2] VMProtect, http://vmpsoft.com/, VMProtect Software.
[3] R. Rolles, “Unpacking Virtualization Obfuscators,” in Proceedings of the 3rd USENIX Conference on Offensive Technologies, ser. WOOT’09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1.
[4] L. Bohne, “Pandora’s Bochs: Automated Unpacking of Malware,” 2008.

[5] D. Schellekens, B. Wyseur, and B. Preneel, “Remote Attestation on Legacy Operating Systems with Trusted Platform Modules,” Sci. Comput. Program., vol. 74, no. 1-2, pp. 13–22, Dec. 2008.

[6] S. Pearson, Trusted Computing Platforms: TCPA Technology in Context. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[7] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, “A Trusted Open Platform,” Computer, vol. 36, no. 7, pp. 55–62, Jul. 2003.

[8] C. Tarnovsky, “Semiconductor Security Awareness Today and yesterday,” in Blackhat, 2010. [Online]. Available: https://www.youtube.com/watch?v=WXX00tRKOlw

[9] C. Tarnovsky, “Attacking TPM part two,” in Defcon, 2012. [Online]. Available: https://www.youtube.com/watch?v=Ed 9p7E4jIE

[10] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, “Truly-Protect: An Efficient VM-Based Software Protection,” Systems Journal, IEEE, vol. 7, no. 3, pp. 455–466, 2013.

[11] M. Kiperberg and N. J. Zaidenberg, “Efficient Remote Authentication,” in The Journal of Information Warfare, vol. 12, no. 3, 2013.

[12] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer’s Manual, 2007, vol. 3.

[13] “AMD64 Architecture Programmer’s Manual Volume 2: System Programming,” AMD, 2010.

[14] M. Pietrek, “An in-depth look into the Win32 portable executable file format,” in MSDN Mag. 17, 2, 2002, pp. 80–90.

[15] E. Youngdale, “Kernel korner: The elf object file format by dissection,” Linux Journal, vol. 1995, no. 13es, p. 15, 1995.

[16] G. J. Popek and R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures,” Commun. ACM, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[17] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, “Bitvisor: A thin hypervisor for enforcing i/o device security,” in Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE ’09. New York, NY, USA: ACM, 2009, pp. 121–130.

[18] Y. Chubachi, T. Shinagawa, and K. Kato, “Hypervisor-based Prevention of Persistent Rootkits,” in Proceedings of the 2010 ACM Symposium on Applied Computing, ser. SAC ’10. New York, NY, USA: ACM, 2010, pp. 214–220.

[19] R. Kennell and L. H. Jamieson, “Establishing the Genuinity of Remote Computer Systems,” in Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12, ser. SSYM’03. Berkeley, CA, USA: USENIX Association, 2003, pp. 21–21.

[20] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, “Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems,” in Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, ser. SOSP ’05. New York, NY, USA: ACM, 2005, pp. 1–16.

[21] Q. Yan, J. Han, Y. Li, R. H. Deng, and T. Li, “A software-based root-of-trust primitive on multicore platforms,” in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ser. ASIACCS ’11. New York, NY, USA: ACM, 2011, pp. 334–343.

[22] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, “SWATT: softWare-based attestation for embedded devices,” in Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on, May 2004, pp. 272–282.

[23] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, “On the Difficulty of Software-based Attestation of Embedded Devices,” in Proceedings of the 16th ACM Conference on Computer and Communications Security, ser. CCS ’09. New York, NY, USA: ACM, 2009, pp. 400–409.

[24] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, “Scuba: Secure code update by attestation in sensor networks,” in Proceedings of the 5th ACM Workshop on Wireless Security, ser. WiSe ’06. New York, NY, USA: ACM, 2006, pp. 85–94.

[25] Y. Yang, X. Wang, S. Zhu, and G. Cao, “Distributed software-based attestation for node compromise detection in sensor networks,” in Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems, ser. SRDS ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 219–230.

[26] D. Ionescu, “Microsoft bans up to one million users from xbox live,” PC World, Tech. Rep., 2009. [Online]. Available: http://www.pcworld.com/article/182010/xbox users banned.html

[27] Sony, “Information on banned accounts and consoles,” Sony consumer electronics, Tech. Rep., accessed on may 2015. [Online]. Available: https://support.us.playstation.com/app/answers/detail/a id/ 1260/~/information-on-banned-accounts-and-consoles
[28] Brian, “Nintendo starting to ban pirates from online services on 3ds,” Nintendo everything, Tech. Rep., 2015. [Online]. Available: http://nintendoeverything.com/ nintendo-starting-to-ban-pirates-from-online-services-on-3ds
[29] Wikipedia, “An analysis of proposed attacks against genuinity tests,” Tech. Rep., accessed on May 2015. [Online]. Available: http://en.wikipedia.org/wiki/Warden (software)
[30] M. Kiperberg, A. Resh, and N. J. Zaidenberg, “Remote Attestation of Software and Execution-Environment in Modern Machines,” in CSCloud, 2015.
[31] M. Larabel and M. Tippett, “Phoronix test suite,” Phoronix Media, Tech. Rep., accessed on June 2, 2016. [Online]. Available: http://www.phoronix-test-suite.com/
[32] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, C. Rozas, "Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave," Proceedings of the Hardware and Architectural Support for Security and Privacy, Seoul, Republic of Korea: ACM, 2016, pp. 1-9
[33] M Kiperberg, N. J. Zaidenberg “Efficient Remote authentication” Journal of information warfare October 2013
[34] A. Resh, N. J. Zaidenberg “Can keys be hidden inside the CPU on modern windows host” ECIW 2013 pages 231-235

PVI

HYPERVISOR-ASSISTED DYNAMIC MALWARE ANALYSIS

by

R. Leon, M. Kiperberg, A.A. Leon Zabag, N.J. Zaidenberg

ACM Transactions on Privacy and Security (TOPS), Submitted