JYVÄSKYLÄ STUDIES IN COMPUTING
255

Amit Resh

# Enforcing Trust for Execution-Protection in Modern Environments

JYVÄSKYLÄN YLIOPISTO

JYVÄSKYLÄ STUDIES IN COMPUTING 255

Amit Resh

# Enforcing Trust for Execution-Protection in Modern Environments

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi yliopiston Agora-rakennuksen auditoriossa 3
joulukuun 19. päivänä 2016 kello 12.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
in building Agora, auditorium 3, on December 19, 2016 at 12 o'clock noon.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2016

# Enforcing Trust for Execution-Protection in Modern Environments

JYVÄSKYLÄ STUDIES IN COMPUTING 255

Amit Resh

# Enforcing Trust for Execution-Protection in Modern Environments

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2016

Editors
Timo Männikkö
Department of Mathematical Information Technology, University of Jyväskylä
Pekka Olsbo, Ville Korkiakangas
Publishing Unit, University Library of Jyväskylä

URN:ISBN:978-951-39-6887-8
ISBN 978-951-39-6887-8 (PDF)

ISBN 978-951-39-6886-1 (nid.)
ISSN 1456-5390



Jyväskylä University Printing House, Jyväskylä 2016

## ABSTRACT

Resh, Amit
Enforcing Trust for Execution-Protection in Modern Environments
Jyväskylä: University of Jyväskylä, 2016, 98 p. (+included articles)
(Jyväskylä Studies in Computing
ISSN 1456-5390; 255)
ISBN 978-951-39-6886-1 (nid.)
ISBN 978-951-39-6887-8 (PDF)
Finnish summary
Diss.

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.

Modern processors contain hardware virtualization capabilities that support implementation of hypervisors for the purpose of managing multiple Virtual-Machines (VMs) on a single computer platform. The facilities provided by hardware virtualization grant the hypervisor control of the hardware platform at an effective privilege level that supersedes the OS.

The purpose of this work is to research and develop a methodology based on a thin-hypervisor that exploits the virtues of hardware virtualization for the purpose of protecting a computer system against malicious penetration. To successfully accomplish this, the thin-hypervisor must be guaranteed to be trusted, with respect to its instructions its configuration structures and its true control over the hardware platform. Moreover, it must be able to protect itself indefinitely from subversion. The methodology presented here describes the means to establish a trusted thin-hypervisor and demonstrates how it may be exercised to restrict code execution exclusively to pre-signed, whitelisted, software.

This methodology provides resistance to most APT attack vectors, including those based on zero-day vulnerabilities that may slip under behavioral-pattern radars.

Keywords: cyber protection, APT prevention, hypervisor, thin-hypervisor, virtualization, attestation, trusted computing, whitelisting,

**Author**

Amit Resh
Department of Mathematical Information Technology
University of Jyväskylä
Finland

**Supervisors**

Professor Pekka Neittaanmäki
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Doctor Nezer Zaidenberg
Department of Mathematical Information Technology
University of Jyväskylä
Finland

**Reviewers**

Adjunct Prof., Dr. Jarmo Siltanen
Director Institute of Information Technology
JAMK University of Applied Sciences
Finland

Dr. Nethanel Gelernter
School of Computer Science
The College of Management Academic Studies
Israel

**Opponent**

Adjunct Prof., Dr. Jyri Rajamäki
Laurea University of Applied Sciences
Finland

## PREFACE

This research began with the ***TrulyProtect*** project, funded by TEKES – the Finnish funding agency for Technology and Innovation and the University of Jyväskylä. The project was launched as an effort to achieve commercialization of academic ideas. The author personally joined the ***TrulyProtect*** project in December 2012. Since, he has been a chief member and contributor to the project team, as well as a major driver of its research and implementation effort. When setting out to explore new alternatives for creating trusted platforms and software protection schemes, the ***TrulyProtect*** team had only put forth vague goals and could not entirely foresee the final destination, as it needed to navigate uncharted waters. Eventually, the methodology crystalized and the research bore fruit, as described in this work. Funding for the ***TrulyProtect*** project supported the team's efforts throughout September 2014, after which continued research was based on individual grants and scholarships.

## ACKNOWLEDGEMENTS

I would like to thank my supervisors, Prof. Pekka Neittaanmäki for his guidance, counseling and assistance in navigating the University terrain, as well as Dr. Nezer Zaidenberg, for his constant support, advice and generally nudging me in the right direction.

Likewise, I would like to express my gratitude to the external reviewers, Dr. Jarmo Siltanen and Dr. Nethanel Gelernter, who thoroughly read my work and provided their valuable comments and insights. I express my thanks, as well, to Dr. Jyri Rajamäki, who generously agreed to act as my opponent.

I would also like to thank the TrulyProtect team, Roee Leon and Asaf Algawi for their support in this research project and assistance in co-authoring some of the joint publications included here, and especially so to Dr. Michael Kiperberg, who served as my main research colleague and with whom I could scrutinize and bounce-off ideas and speculations on an almost daily basis.

I am also indebted to the support of the Ellen and Artturi Nyyssönen Foundation, the COMAS Graduate School and the Department of Mathematical Information Technology, which provided financial assistance for this research, for which I am extremely grateful.

I also owe a great thanks to my beloved wife, Racheli, for enduring this effort and me during the seemingly endless months and years, for her love, understanding, moral support and devotion.

Finally, I would like to thank my children Eyal, May and Eran, Racheli's daughters Dana and Noa, as well as my parents Dr. Nura Resh and Dr. Michael Resh, who also provided important comments to my drafts, for their wholehearted encouragement, support and approval.

Jyväskylä
December 2016
Amit Resh

## LIST OF FIGURES

## LIST OF TABLES

## CONTENTS

## LIST OF INCLUDED ARTICLES

PI Resh, A.; Kiperberg, M.; Leon, R.; Preventing Execution of Unauthorized Native-Code Software. To be published in: *JDCTA, International Journal of Digital Contents Technology and its Applications*, 2016.

PII Resh, A.; Kiperberg, M.; Leon, R.; Zaidenberg, N.J.. System for Executing Encrypted Native Programs. To be published in: *JDCTA, International Journal of Digital Contents Technology and its Applications*, 2016.

PIII Kiperberg, M.; Resh, A.; Zaidenberg, N.J.. Remote Attestation of Software and Execution-Environment in Modern Machines. *The 2nd IEEE International Conference on Cyber Security and Cloud Computing*, 2015.

PIV Zaidenberg, N.J.; Resh, A.. Timing and Side Channel Attacks. *Cyber Security: Analytics, Technology and Automation, vol. 78, pp. 183-194*, 2015.

PV Zaidenberg, N.J.; Neittaanmäki, P.; Kiperberg, M.; Resh, A.. Trusted Computing and DRM. *Cyber Security: Analytics, Technology and Automation, vol. 78, pp. 205-212,* 2015.

PVI Resh, A.; Zaidenberg, N.J.. Can keys be hidden inside the CPU on modern Windows host. *ECIW 12th European Conference on Information Warfare and Security, Jyväskylä*, 2013.

PVII Kiperberg, M.; Resh, A.; Algawi, A.; Zaidenberg, N.J.. System for Executing Encrypted Java Programs. *38th IEEE Symposium on Security and Privacy (IEEE S&P 2017),* Submitted.

PVIII Kiperberg, M.; Resh, A.; Algawi, A.; Zaidenberg, N.J.. System for Executing Encrypted Java Programs. *3rd International Conference on Information Systems Security and Privacy (ICISSP 2017),* 2017.

# 1 INTRODUCTION

## 1.1 Modern System Execution Vulnerabilities

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.

An assortment of recent cases bear witness to this escalating and dire problem:

- **Target** (Nov 2013): Target is one of the largest discount retailers in USA, second only to Walmart. Malware designed to capture the details of swiped credit cards was installed in Target's payment server just prior to Thanksgiving sales. Roughly 40 million customer credit cards were abducted [1].
- **JP Morgan Chase** (Oct 2014): With assets surpassing $2 trillion, it is the largest bank in the USA and one of largest in the world. Four hackers penetrated the bank servers and obtained illegal access to over 80 million customer accounts, thereby reaping over $100M using these for online gambling, phishing and money laundering to name only a few [2].
- **Anthem** (Feb 2015): The largest health insurance company in the Blue-cross Blue-Shield association. In Feb 2015, Anthem reported that its database had been breached and 80M current and past patient credentials and medical data had been exposed [3].
- **Premera Health** (Mar 2015): Premera Health is a large, non-profit, Blue-cross Blue-shield health insurance company. The company reported that hackers broke into its database exposing 11 million customer records [4].
- **Ashely Madison** (July 2015): Ashely Madison is an online dating service geared towards married people looking for an ex-marital relationship. In July 2015 a hacker group calling itself "The Impact Team" hacked Ashely

> Madison's computers and stole its entire user base. The group tried to blackmail the site into shutting down and ended up leaking 25 Gigabytes of material when their demands were not met. The data breach caused an immense impact on the lives of the people involved, including two suicides linked to the event [5].

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, SophosLabs [6] 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 hijacked 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 Host 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 executable code. SophosLabs identify several common malware payloads: Zbot(Zeus) – 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 penetrate a system 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 un-privileged code exploits a system vulnerability (usually in the O/S) 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 file system on disk – 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 screen-shots or even achieve full remote-control of the system.

## 1.2 Creating Trust in a Remote System

In general, achieving ***Trust*** in remote computer systems should be interpreted as generating a specific instance or object, which can be trusted and relied upon to act in a predetermined way under all circumstances [7]. In general, ***Trust*** encompasses validated software combined with some secret data known only to that software coupled with a methodology that assures protection against subversion and/or modification of the secret data.

Therefore, ***Trust*** must be created, validated and then (indefinitely) sustained.

The problem of remote authentication, determining whether a remote computer system is running the correct software version, is well known [8] [9] [10] [11]. Equipped with a remote authentication method, a service provider can prevent 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 [12] [13] and only authenticated bank terminals can be allowed to fetch records from the bank database [14].

The research in this area can be divided into two major branches: hardware assisted authentication [15] [16] and software-only authentication [8] [9]. While in theory, hardware assisted authentication may provide more conclusive results regarding the authenticity of a remote machine, in practice the hardware fails to provide additional security due to inappropriate designs of currently available operating systems [17].

Hardware assisted authentication uses an external hardware component, such as a Trusted Platform Module (TPM), to compute a cryptographic hash of the computer's hardware and software configuration to attest it. Frequently the TPM is used as the root of a chain of trust [18]. The TPM measures the authenticity of the BIOS. The BIOS then measures the authenticity of the boot loader and so on. Unfortunately, all common modern operating systems (e.g. Linux, Windows, OS X) allow the user to load drivers for execution with the same privileges as the operating system itself, i.e. ring 0 on x86 and x64 hardware. Malicious or buggy drivers, which are executed with high privileges, allow random code execution and thus make it possible to circumvent the authenticity measurements of the TPM. Physical attacks on TPM were also shown to exist, assuming the assailant has access to the hardware were it is installed [19] [20] [21].

System-wide 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. These 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.

## 1.3 Methods of Obfuscation

One way of preventing circumvention of software, is by using methods of ***obfuscation*** [22] [23] [24] [25]. The term obfuscation refers to making software instructions difficult for humans to understand by deliberately cluttering the code with useless, confusing pieces of additional software syntax or instructions. Obfuscated code must still be functional, however its goal is to render the code difficult enough to understand and therefore too difficult to reverse-engineer. The assumption is that without properly reverse-engineering software, it is not possible to subvert or circumvent it to the gain of the aggressor. In most cases obfuscation methods attempt to protect two main aspects of malicious attacks: (a) software piracy; and (b) software tampering. Software piracy takes on the form of operating or redistributing software without a license or stealing software intellectual property, such as an algorithm used in a software product. Software tampering involves making changes to existing software, such as circumventing instructions that may check for licensing or by adding instructions that are designed to achieve some malicious activity, such as a virus or any other form of malware that needs to infiltrate the system.

More advanced, software publishers may protect their digital content product by encryption, using a unique key to convert the software code or data to an unreadable format, such that only the owner of the unique key may decrypt the software code. Such protection, however, is only effective when the unique key is kept secured and unreachable to an adversary. This reduces the security issue to that of securing the key. Since the software must also function properly in its untampered form, it must have the key available, leading to the necessity to obfuscate the key and the routines that make use of the key.

It has been shown that obfuscated software code can be invariably broken by hackers, specifically since its content must still be readable to properly function [26] [27] [28] [29] [30]. Hackers equipped with the proper tools, such as disassemblers, logic analyzers, tools for static and dynamic analysis coupled with patience and dedication, have cracked even the most cunning software obfuscations methods.

## 1.4 Security by Design

Security by design [31] [32] proposes an alternative to methods of obfuscation, which propagate security by obscurity. Rather than attempting to secure a system ***after*** its implementation, by hiding or obfuscating its critical elements, security by design addresses the issues of security as part of the system design ***before*** its implementation. Using this approach [33], system design inherently encompasses active security techniques, vulnerability elimination and built-in resistance to attack adhering to best-practices. Security by design steers away from relying on secret operations, obscurity or obfuscation techniques to achieve security. Revealing the security design openly, without compromising its security, often leads to the best security by design methodologies. The reason for this is twofold:

(a) System security does not rely on chance and is not dependent on as adversary's capability to investigate or stumble upon a secret. Security is fundamental to the methodology, where it can be shown that knowing its inner workings does not compromise it; and

(b) The security methodology is open to all, for scrutinizing and peer-review, leading to flaws, if any, being exposed and amended.

## 1.5 Overview of the Proposed Methodology

Protecting computer systems from malicious code, malware and data-breaches in an impregnable manner, must be based on foundations of "security by design". One of the key points in achieving this is successfully creating ***trusted components*** on the target system. The trusted components are comprised of software and its configuration data. In this context, setting up ***trusted software***

on a computer system should be interpreted as software that can be guaranteed to perform in a predetermined manner, whose code contents are validated, that can protect its internal assets (code and data) from subversion and that can perform unique activities (such as cryptographic computations) that prove its authenticity at any time.

### 1.5.1 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.

Furthermore, it is assumed that an adversary cannot obstruct the operation of a root (primary) hypervisor based on hardware virtualization. Nor can an adversary obstruct the protected mechanisms of SLAT (secondary level address translation) (i.e., EPT) and IOMMU that operate at a privilege that is higher than the OS while 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.

### 1.5.2 Proposed Methodology

A general overview of the proposed methodology is presented here which shall be elaborated in detail in the following chapters.

Computer systems that run application software are normally managed by Operating Systems (OS). The OS manages all hardware resources, schedules software for execution and provides hardware-oriented services [34] [35] to the applications that run above it. Since the OS is an intermediary between the hardware and the software, it must have full control over the hardware resources to properly manage and allocate them to the software applications in such a manner so as to maintain system integrity. As such, OS software routines must be given a higher privilege level than the application software. The elevated privilege assures that application software cannot circumvent the OS in accessing the hardware or software resources, which must be exclusively controlled by the OS. For this purpose, the central core of the OS, called the Kernel [36] [34], is composed of a group of routines that operate at an elevated privilege level. The Kernel routines carry out all the hardware and critical software management task. In modern processors, privilege level is enforced by a set of instructions that can only be performed when the processor is put in a high privilege mode. The OS configures the system so that only the kernel

routines operate at the higher privilege mode, while all other software applications operate at a lower privilege level and thus require kernel routine assistance in utilizing hardware resources.

A critical security restriction, that must be enforced, is ensuring that malware does not infiltrate the OS kernel and can thus execute at an elevated privilege level, consequently allowing it to obtain full control of the computer system. However, this is easier said than done. Due to the size and complexity of operating systems, hackers and malware programmers are continuously finding vulnerabilities that are exploited to allow malicious software to gain a high privilege level and compromise the system. Therefore, the methodology proposed herein suggests the use of a software component having a higher privilege than the OS. Thus, it can be used to manage system security, even in light of the possibility that malware has infiltrated the kernel and achieved OS privilege level execution rights.

The embodiment of a software component with privileges higher than the OS has been realized in the form of a hypervisor [37] [38] and is used to manage several operating systems on a single hardware platform. Hypervisors, first introduced in mainframe computers in the 1960s, now utilize hardware virtualization [39] that is available in most modern processors.

Rather than use a hypervisor as a multi-operating-system manager, it is proposed to utilize its elevated privilege to manage and monitor a systems security. However, to achieve that, the hypervisor itself must be guaranteed to be ***trusted*** and completely free of malware and vulnerability to subversion. Two main aspects of hardware virtualization technology position the hypervisor as a favorable candidate for a software component that can be remotely trusted to be safe. First, a minimal hypervisor can be created and therefore with minimal complexity, making it easier to verify. And second, once a verified (and authenticated) hypervisor is in control of a hardware platform it is potentially able to resist all attempts of subversion.

The proposed methodology, described in this work, proposes to run a minimal hypervisor for the sole purpose of securing a computer system. Furthermore, an ***attestation*** procedure [8] [40], governed by an external server system, is used to ensure that the hypervisor is trustworthy by authenticating and validating its contents. The attestation procedure will serve to guarantee that a hypervisor, after taking control of a remote system, is trustworthy and can be safely assumed to maintain its trustworthiness so-long as the system remains in operation. Once trustworthiness is established, the attestation server can transfer secret information directly to the trusted hypervisor in a safe manner. The secret information can then be used by the hypervisor to carry out cryptographic operations allowing it to further communicate, in confidence, with the attestation server. It can also receive, interpret and validate additional encrypted information, utilizing that to augment maintenance of cyber-security in a computer system.

### 1.5.3 Existing Methodologies Evaluation

In 2016, Microsoft added a similar methodology to Windows 10, called ***Device Guard*** [41]. Device Guard is a group of features designed to take advantage of hardware-virtualization, SLAT and IOMMU to protect systems against unsigned applications, malware and APTs. Microsoft calls this technology ***Virtualization Based Security*** (VBS). The technology Microsoft offers is based on Microsoft's own Hyper-V, which is a full blown hypervisor that is integrated into Windows OS.

To take advantage of Hyper-V it must be booted before the Windows OS. Similar to the methodology proposed in this thesis, when Device Guard is employed, the hypervisor verifies code integrity in both Kernel and User mode applications and manages memory access rights based on SLAT. Microsoft provides tools that sign applications (whitelisting) and when Device Guard is active, verifies signatures with components called ***CCI*** (Configurable Code Integrity) and HVCI (Hypervisor Code Integrity).

Since Hyper-V is a full blown hypervisor, its attack surface is relatively large, which leaves an opening for exploitation attacks [42]. Hyper-V, which is booted before the OS, is not inherently secure and thus cannot be trusted. Therefore, potential attacks on the integrity of Hyper-V may be attempted to circumvent Device Guard. A secure boot, based on TPM, could be employed, however besides the added TPM hardware requirement, TPM has been broken, as mentioned above [19] [20] [21].

Furthermore, since the technology is based on Hyper-V, it is applicable only to systems running Microsoft operating systems.

## 1.6 Research Contribution

A methodology and system that achieve a strong system-wide protection against execution of a wide array of unauthorized code penetrations is proposed and studied. The research 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 lean thin-hypervisor proposed provides for an extremely small surface of attack. The attestation procedure described provides a software-only solution that ensures the hypervisor can be trusted and contains safeguarded secret key material.

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 hypervisor protection (see chap. 4.5.1), then shifting dynamically into protected mode to run office applications that

warrant extensive security. Applications that execute in protected mode shall be protected and isolated from any 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.

The lean thin-hypervisor design is extremely apt to porting to other operating systems. Currently it is operating on Windows, IOS and Linux with ports to ARM/Android underway. Furthermore, since the proposed thin-hypervisor is unrelated to a specific operating system, it may be used as a hosting hypervisor in a nested-hypervisor environment to provide application execution protection for applications running on multiple operating systems on the same computer system.

## 1.7 Author Contribution

The author is a major team member of the TrulyProtect research team, which was financed by the Finnish agency TEKES and the University of Jyväskylä. The author significantly contributed, together with the other team-members, to the research project in conceiving the ideas and developing the theory and the framework behind setting-up a trusted thin-hypervisor and its remote-attestation for the purpose of securing a computer system against malicious software attacks. The author also substantially contributed to writing the software that implements the proposed methodologies that are studied under this research.

Summaries of the included articles along with the author's contribution to each are detailed in chapter 6.

# 2 USING A HYPERVISOR TO ENFORCE TRUST

## 2.1 Hypervisors and Hardware-Assisted Virtualization

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.

Hypervisors have been in use as early as the '60s on IBM mainframe computers [43]. After 2005 Intel and AMD have added hardware support in the form of virtualization extension instructions that are an extension to the x86 instruction set architecture, allowing isolation of multiple operating systems efficiently, thus facilitating the construction of virtual machine monitors (Hypervisors) [44] [45]. Note that previously, construction of virtual machine monitors involved binary instrumentation and required modification in the code of the hosted operating systems.

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 are further categorized as: type-1 [46] (or bare metal) and type-2 hypervisors [47].

A type-1 hypervisor executes independently and directly over the system hardware. The OS of the Guests run above the hypervisor, in effect decoupled from the system hardware by the hypervisor.

A type-2 hypervisor executes above a cooperating OS, where Guests run atop the hypervisor. This type of hypervisor uses the cooperating OS as a means to access and manage hardware resources.

In order to support multiple OS guests, a type-1 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 depicts a virtualized system featuring a hypervisor that manages two Virtual-Machines (VMs), each running an operating system that manages its user applications. The hypervisor runs at 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 systems handle these conditions by requesting services from the underlying hardware. The hypervisor is configured to intercept all those requests and handle them according to its policies.

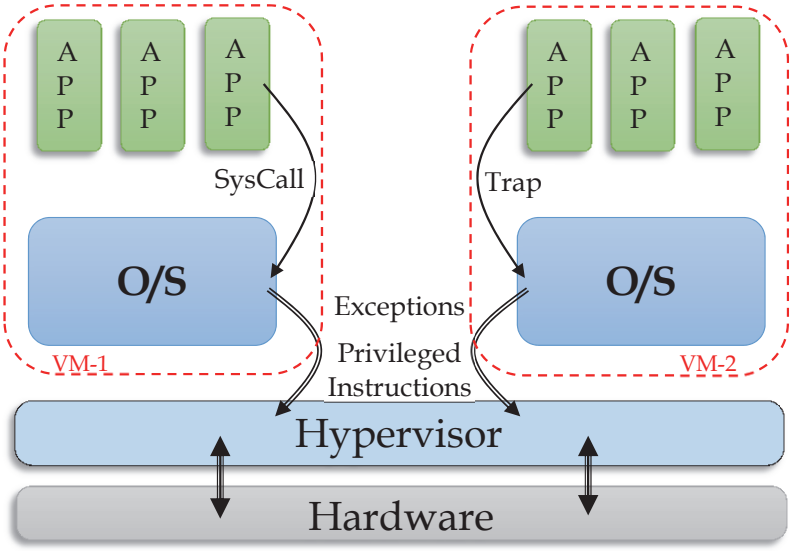


FIGURE 1 Virtualized system with 2 Virtual-Machines (VM). Each VM is a stack comprised of an OS with applications running over it. Applications utilize the OS by making system calls and Trap intercepts. Each OS believes to be running over the hardware platform, however OS requests from the hardware are intercepted and managed by the hypervisor, which maintains isolation between the VMs.

To intercept all OS hardware access, hypervisors are 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 VM 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 type-1 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, type-1 hypervisors, which control the underlying hardware platform, also providing a very good fit to serve as software security facilitators.

Hypervisors have been previously used to secure systems. For example, the Software-Privacy Preserving Platform (SP3) [48] 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.

## 2.2 The Thin-Hypervisor

Our research project proposes to use a type-1 hypervisor environment for securing a single Guest stack. Rather than wholly virtualizing the hardware platform, a special breed of hypervisor, called a thin-hypervisor [49] [50], is used. 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. The thin-hypervisor does not control most of the OS interaction with the hardware, therefore multiple OSs are not supported. However, system performance is kept at an optimum. Additionally, the thin-hypervisor runs at 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 service from the underlying hardware. The thin-hypervisor intercepts only a few of those requests, while remaining transparent to all others, which are thus serviced directly by the hardware.

Thin-hypervisors have been previously used for security purposes. For example, TrustVisor [51] 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.

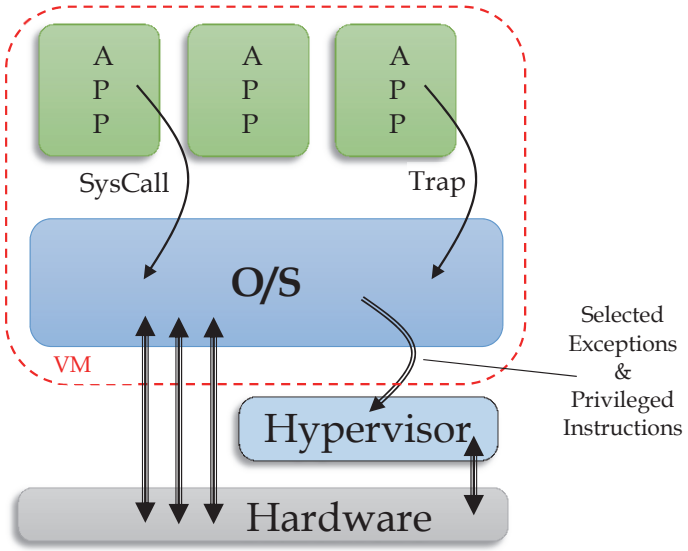


FIGURE 2 Thin-hypervisor securing a single VM stack. The thin-hypervisor only virtualizes a select subset of the OS's hardware requests. Most OS operations are executed directly by the hardware.

A thin-hypervisor facilitates a secure environment by:

(a) Setting aside portions of memory that can be accessed only when the CPU is in Host mode
(b) Storing cryptographic key material in privileged registers, and
(c) Intercepting privileged instructions that may compromise its protected memory or key material

A thin-hypervisor is also 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. This serves to significantly reduce the threat surface that needs to be protected.

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.

A correctly configured and active thin-hypervisor can effectively protect the secret key-material, after it is safely stored in privileged registers. 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 approach to solving these issues is comprised of the following principles:

1. While the thin-hypervisor is not active, the secret key material shall not be stored anywhere in the system
2. When setting up a thin-hypervisor, an external system shall be used to verify that the thin-hypervisor has control over the underlying hardware
3. 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 attestation server system and equipping it with the facilities to verify that a thin-hypervisor has been properly setup and configured on a remote system, such that a trusted environment is primed and can accept secret material, as well as keep it secret.

## 2.3 Attestation of a Remote Hypervisor Activation

Hypervisors can be nested. In other words, a hypervisor's Guest can itself be a virtualized system embodying a VMM (Host) and VMs (Guests) [52]. In order to support such a configuration the outermost surrounding hypervisor needs to specifically support nested-virtualization.

We introduce the concept of a ***Root-Hypervisor***: A Root-Hypervisor is the distinct hypervisor that has the ultimate control over the hardware platform. A Root-Hypervisor does not have to support nested-virtualization. However, in a nested virtualization environment the surrounding hypervisor, which supports and manages nesting is invariably the Root-Hypervisor.

Hypervisor activation can be part of the system boot process or can occur ***after*** an OS is already active. This is achieved by capturing the OS instance, taking (virtualization) control of the system and instantiating the OS instance as a Guest (VM). Such a rootkit driver, dubbed "***Blue-Pill***" was suggested in 2006 by Joana Rutkowska [53] and independently by King and Chen [54].

Configuring and installing a hypervisor can only occur while the system is in Kernel-mode, since hypervisor instructions are all privileged instructions. Therefore, when launching a "*Blue-Pill*" style hypervisor, under the supervision of an operating system, the hypervisor configuration and installation functions must be implemented in a kernel-mode system software driver. We call this the ***HDriver*** (Hypervisor-Driver). The target system must have the HDriver installed and registered in order to successfully launch the hypervisor installation.

Once a Root-Hypervisor has taken control of the system, by becoming a VMM host, it is capable of intercepting all further hardware events. This includes the aforementioned capability to manage and support the upbringing of a nested-hypervisor by one of its Guest VMs. Furthermore, the Root-Hypervisor can take measures to conceal its existence. For example, by falsifying results of intercepted instructions that probe the hardware. This type of behavior has been suggested as an approach for hackers implementing hypervisor-based malware [55] [56].

It is self-evident that the Root-Hypervisor needs to be the **first** hypervisor to take control of the system. Being the first hypervisor to virtualize the system secures the opportunity to intercept hardware events as well as fully control the intercept activity of nested hypervisors.

When attempting to use a hypervisor as the base foundation for enforcing trust in a computer system it is unequivocally essential to determine that the hypervisor is a ***Root-Hypervisor***. In other words that it is the first hypervisor to virtualize the system, and thus has ultimate control of the hardware platform. Given the disingenuous capabilities of a hardware-virtualization based hypervisor, this task as not as simple as may initially appear [57] [58].

However, verifying that the hypervisor is a Root-Hypervisor is not the only concern in establishing trustworthiness. Granted, once the hypervisor is in control and validated as a Root-Hypervisor, it may be safe to assume that it can be utilized to enforce trust, as well as protect itself, as discussed in the paragraph below. However, during the process of *establishing* the hypervisor's control, even assuming it is the first hypervisor, it is critically exposed to subversion. The main concerns are the possibility of a malicious software making changes to the hypervisor's code or data-structures just ***before*** it takes control. These concerns are especially acute when the hypervisor is launched while an OS is already prevailing, since the potential of its being infected by malicious code is eminent. Furthermore, modern computer CPUs consist of more than a single processing unit, organized in a hierarchy of ***cores*** and ***logical processors***. With multiple cores the system can execute several programs in parallel. This imposes additional verification requirements, since it is potentially possible for malicious code running on one core to undermine the process of establishing the hypervisor on another. Moreover, the system can be regarded as trustworthy only if a root-hypervisor is successfully instantiated and verified on **all** the existing cores.

***Attestation*** [59] [60] is the collective effort of verifying and validating that the hypervisor is a root-hypervisor, properly installed on all cores, that has not been subverted in any way. Namely it establishes a ***trusted platform*** in a target computer system.

One of the main instruments to facilitate attestation is ***time-measurements***. The reason for this is the execution overhead associated with virtualization intercepts. For example, an attestation procedure may measure the time it takes to complete a pre-known procedure and attempt to determine whether intercepts occurred and accordingly consumed time. However, timing measurements cannot be conducted by the system being attested, since requests to fetch time readings from the hardware can be intercepted and the results falsified. Hence, an external system must be involved in the attestation process. We use an ***Attestation Server*** for this purpose and perform a ***Remote-Attestation Procedure***. The attestation server communicates with the attestation target, sends it a ***challenge*** in the form of executable instructions and measures the turn-around response time. In this case, the time measurement can be considered objective.

## 2.4 Protecting a Thin-Hypervisor that Enforces Trust

We assume at this point that a thin-hypervisor is brought up on a computer system and it is successfully attested by an attestation-server, therefore it can be trusted. In the next chapter we will explain and demonstrate that following a successful attestation procedure we can assume the following:

- The hypervisor is a Root-hypervisor, i.e., it is the first hypervisor to be launched in the computer system and it has control of the underlying hardware
- The hypervisor code contents are verified and authenticated, i.e., instructions that will execute in host mode have not been subverted, therefore the hypervisor can be trusted to perform as intended
- The hypervisor contains secret material, received in confidence from the attestation-server, during the attestation procedure. The secret material is known only to the hypervisor and the attestation-server

A significant part of the hypervisor configuration and routines must be dedicated to enforcing and maintaining trust ***after*** the attestation has validated the initial conditions of trust. To accommodate, the hypervisor's surface of attack must be considered and appropriate means need to be deployed to provide ample resistance. The following major aspects of hypervisor trust maintenance are reviewed, as examples:

### 2.4.1 Secret Material Storage

The hypervisor receives secret material in confidence from the attestation-server. The secret material is a basis for cryptographic operations, such as decryption or signing, which the hypervisor carries out as part of the general security-scheme implemented by the hypervisor. Naturally, this secret material must be kept out of reach and is a potential target for an adversary wishing to subvert the hypervisor functions. To overcome this, the hypervisor stores the secret information in privileged registers, as well as configures VM exit intercepts on access attempts to the privileged registers. See section 25.1.3 in [44]. Any attempt to access these registers will be intercepted by the hypervisor, which will either ignore the access request or report an invalid result.

### 2.4.2 Protecting Hypervisor Configuration-Structures

Hypervisor configurations are stored in dedicated data-structures. For example, VMCS in Intel, section 24.1 [44] and VMCB in AMD, section 15.5.1 [61]. The contents of these repositories must be vigorously defended from illicit access. Otherwise an adversary might make changes that will eventually subvert the hypervisor. For example, she may replace the address setting of the intercept handler, and thus, when a VM exit intercept occurs, the adversary's intercept routine shall be activated instead of the intended one. To subjugate this threat, the hypervisor takes advantage of a mechanism called Second Level Address Translation (SLAT). See Intel EPT chapter 28 of [44] and AMD RVI chapter 15.25 [61]. SLAT is discussed later in section 4.1. At this point we only mention that SLAT allows a hypervisor complete control over memory access-rights. Subject to this, the hypervisor configures the SLAT to disallow any access to the memory pages that contain it configurations. In effect, this memory does not exist outside host mode and therefore cannot be accessed.

### 2.4.3 Using Intel VT-d and AMD-Vi (IOMMU)

While SLAT provides an immaculate solution to protect against memory access performed by software running on one of the core processors, I/O device DMA transfers provide an alternate route to access memory. Virtual to physical address translation during CPU memory transfers is managed by the MMU (Memory Management Unit), which implements SLAT when hardware virtualization prevails. On the other hand, I/O device DMA allows devices to access memory directly. An adversary may potentially attempt to affect critical memory sections protected by the hypervisor with SLAT, using DMA or RDMA (Remote DMA), thus subverting the hypervisor and penetrating the system. Fortunately, Intel and AMD have implemented solutions for this issue to enhance hypervisor performance. The motivation for this was to allow a hypervisor means to configure I/O device DMA access that ensures VM separation without resorting to hypervisor intercepts on each I/O access. Intel have added the IOMMU and VT-d technology, called "Virtualization

Technology for Directed I/O" [62] and AMD have added AMD-Vi technology "AMD I/O Virtualization Technology (IOMMU) Specification" [63]. AMD-Vi and Intel VT-d technology provide facilities to configure the IOMMU to remap DMA addressing. The configuration is used to map separate VM devices to individual and isolated memory domains. Each VM can then access only its own memory domain and is blocked from other VM domains. To avert potential subversion by accessing memory, the thin-hypervisor configures the IOMMU to remap the (single) VM's I/O DMA access to a memory domain that excludes the same memory pages protected by the SLAT. This procedure ensures that critical pages, whose access is allowed only by the hypervisor, are protected both from memory access as well as DMA and RDMA access. In effect, being completely invisible to the outside world.

### 2.4.4 Secure AES Cryptography

Based on the availability of secret material stored inside the hypervisor and protected in privileged registers, the hypervisor may perform cryptographic operations as part of the general security-scheme. An adversary that is aware of this, may attempt to attain the AES key using side-channel attacks [64] [65]. To avoid this threat the hypervisor uses a hardware implementation of AES, available on modern processors, such as AES-NI (AES New-Instructions) [66]. Furthermore, the implementation of AES cryptography is managed entirely in CPU registers, as opposed to using memory buffers for intermediate results. This combination reduces vulnerability to potential side-channel attacks to virtually non-existent [67] [68].

### 2.4.5 Intercepting Critical Instructions

The thin-hypervisor must also protect its existence. Since malicious code may also penetrate the system and achieve kernel-level execution mode, it is imperative to intercept all privileged instructions that may obstruct the hypervisor's presence. Thus, for example, the ***VMXOFF*** instruction, which causes the system to exit Virtualization mode must be intercepted and ignored. Similarly, ***VMPTRLD***, ***VMPTRST***, ***VMCLEAR***, ***VMWRITE*** and ***VMREAD*** instructions, which access the hypervisor configuration structures must be intercepted and ignored as well.

# 3 REMOTE SOFTWARE ATTESTATION METHODOLOGY

## 3.1 Previous Work

Pioneer [69] is a software-only component designed to provide execution of a remotely authenticated executable on an untrusted and possibly compromised legacy host system. Pioneer is composed of a dispatcher system that is used to manage a challenge-response protocol with the untrusted platform, where an authenticated executable is to be run. The methodology of Pioneer is based on a verification utility, which first establishes itself as a root of trust, by executing code that both checksums itself and verifies that it is running. The verification utility is randomized by receiving a challenge seed from the dispatcher. Once trusted, the verification utility proceeds to authenticate the executable in question. Pioneer is based on two assumptions on the untrusted platform:

(a) It has a single logical processor
(b) It does not contain a virtualization extension

Logical processors multiplicity, which was introduced in modern CPUs, violates the assumptions of Pioneer. The authors propose a remedy for this vulnerability by introducing a data dependency between the different parts of the challenge [70], thus preventing its parallel execution. Pioneer execution on processors with a virtualization extension is discussed in [71]. The authors describe a modification to the original method which allows not only to achieve consistent results on all processors but also to employ intermediate variations to detect virtualized environments.

Kennell and Jamieson proposed a method [8] 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 for modern processor environments [40]. The adaptation solves the security issues that arise from the availability of virtualization extensions and multiplicity of execution units.

## 3.2 Attestation Goals

The design of an attestation procedure for the purpose of establishing trust in a remote multi-processor environment must verify all of the following points:

1. Verify that the hypervisor is a root-hypervisor that has control of the machine, i.e., no emulator or primordial hypervisor already exists in the system
2. Validate the hypervisor contents
3. Verify that the hypervisor that is being attested is the one executing
4. Ensure that all cores/logical-processors are running the attested hypervisor
5. Create a trusted communication channel to the hypervisor
6. Transfer a secret to the trusted hypervisor

Point (1) is required to ensure that the hypervisor being launched and attested is not "fooled" into a false-sense of control by a primordial hypervisor that is already controlling the machine and configures it in a nested-virtualization frame-work. If this scenario could transpire, a malicious thin-hypervisor could potentially grab control of the system before our thin-hypervisor in which case the hypervisor activities could be intercepted and subverted by the malicious root-hypervisor.

Point (2) is essential to validate that the code, which composes the hypervisor, is the intended code and that it can be trusted to behave in a pre-deterministic fashion. Code validation includes the initialization code, to guarantee that the hypervisor configuration data-fields are pre-determined as well and can be trusted. Without this validation the hypervisor would be open to a malicious attack in the form of changing part(s) of its code in a way tailored to reveal its secret or subvert its operation. It can be easily shown that even a single bit changed in the hypervisor's code section would allow completely subverting it. For example, consider changing a significant bit in the address of a routine call that will hurdle the CPU towards a section of malicious code.

Point (3) verifies that the hypervisor that is being attested is actually the one that eventually executes and takes control of the system. This validation is important to counter the possibility that an adversary manages to subvert the

attestation process in a manner that validates the correct hypervisor code. However this code exists in a separate memory buffer but will not be the code to run and control the system after the attestation is complete. If such was the case, the attestation server could be fooled into accepting the remote attestation and giving up the secret to the adversary.

Point (4) is relevant in systems that support multiple processors. In these environments, each processor can execute its own separate hypervisor. Truly controlling the system requires controlling all of the existing processors. Since system memory is shared by all processors, it can be contemplated that if one or more of the existing processors is not configured with the hypervisor, but with a malicious (alternative) hypervisor or no hypervisor at all – system memory contents could be subverted through that processor and thus, trust would be confiscated.

Point (5) is based on the requirement to manage the attestation with a system that is separate from the target system and which can be considered "objective". To support this model, the target system and attesting system must communicate. Communications between the two systems must be secured to avert all possibility to affect the attestation results or allow an adversary to acquire the secret. The security measures here need to account for the possibilities of eavesdropping or man in the middle attacks.

Point (6) is a requirement that facilitates setting up a trusted environment between an ***interested party*** and a remote system that it needs to trust. The secret should generally be interpreted as a cryptographic key. Assuming that the secret is transferred securely to the hypervisor, after it is trusted by virtue of validating all the points above, the interested party can provide the hypervisor information that only it can understand and respond to or receive proof of the hypervisor's hegemony in the remote system.

## 3.3 Hardware side effects

Modern processors manufactured by Intel and AMD provide a facility to count occurrences of side-effect events, internal to the CPU circuitry, called performance events. The main goal behind this feature is to support CPU performance monitoring.

Performance events are defined as internal CPU-circuitry state changes resulting from instruction execution, but not linked directly to the instruction results. For example: cache hit or cache miss events on specific cache memories, such as L1/L2/L3 or the translation lookaside buffer (TLB). The number of possible performance events greatly outnumber the available hardware counter circuits. Therefore, it is possible to dynamically link an available hardware

counter (called a performance counter) to a specific performance event. Once linked, the performance counter counts the number of events that occurred.

In processors manufactured by Intel and AMD, performance counters are realized by a set of model-specific registers. Performance monitoring mechanisms were introduced with the Pentium processor and later evolved with the introduction of the P6 family, Pentium 4, core and all later processors.

In general, some performance mechanisms are architectural. These performance counters are uniformly defined for all processors, while others are non-architectural, meaning they are specific to the micro-architecture and vary between the different processor families. Most processor models are restricted to 2-4 individual performance counters, while the different Xeon-family processors are an exception in their capability to support 9-25 performance counters, depending on the exact model.

## 3.4 Challenges

### 3.4.1 Overview

A challenge in this work, is a piece of native-code, delivered to a remote target system for the purpose of attestation. The challenge is delivered to the target system, where it is executed and produces a result once it completes. The result is transmitted back to the attestation system that originally sent the challenge. The attestation system is responsible to evaluate the result and ultimately decide whether the response can be considered correct, in which case the target can be trusted and the secret information (normally a cryptographic key) may be transmitted back in response.

Challenges need to be devised so-as to calculate a result whose correctness proves all 6 points detailed above and complete within a given timeframe. To achieve this multifaceted goal, the challenge code calculates a hash value of a memory region whose contents includes:

(a) The critical portions of the HDriver (the subject of the attestation)
(b) The challenge itself
(c) A prefabricated virtual-memory page-table (as will be explained below)

It continuously convolute the hash calculation with hardware side-effects that are monitored during the memory scanning process required to calculate the hash. The challenge code, incorporates the intermediate hardware side-effect measurements into its result calculation, as well as governs the calculation flow progress according to the intermediate result value.

### 3.4.2 Challenge Construction

#### *3.4.2.1 Node Network*

A challenge consists of a group of individual ***Nodes***. Each node contains machine code instructions that perform one specific, well defined, operation on an intermediate result. Upon completing its operation the node determines the next node to transfer control to, according to the current intermediate result value. In general, each $Node^{R}$ can transfer control to one of three nodes: $Node^{R}_{[A,B\ or\ C]}$:

(a) If the ***parity*** of the intermediate result is ***even***, control shall be passed to $Noad^{R}_{A}$ – this represents a 50% chance on a random value.

(b) Otherwise, if the current intermediate result is ***positive***, control is transferred to $Noad^{R}_{B}$

(c) Otherwise, it is transferred to $Noad^{R}_{C}$

This represents a 25% chance for each of the two latter cases.

TABLE 1 Node Categories Table

Node operations are generally categorized as:

| | |
|---|---|
| 1. | Hash calculation |
| 2. | Side-effect inducing |
| 3. | Hardware side-effect counter blending |

Each node category contains a group of nodes that carry out an operation pertaining to that category. Nodes that belong to the "***hash calculation***" category shall read the next word from memory and add it to an on-going hash calculation. These nodes shall also progress to the next word in the scanned memory space, as well as determine if the entire memory region was completely scanned, in which case the challenge needs to terminate.

Nodes belonging to the "***side-effect inducing***" category perform an operation that creates a significantly different side-effect on a bare-metal computer system as opposed to a virtualized system.

Nodes belonging to the last category, "***hardware side-effect counter blending***", convolute the intermediate result with one of the side-effect counter values currently monitoring a system hardware side-effect.

Nodes are also subdivided into groups that can be supported on a given CPU architecture. Since more advanced CPUs are normally backwards compatible, nodes that are supported on a certain CPU architecture will usually also be supported on all more advanced architectures.

When constructing a challenge, nodes are selected from a pool of available nodes that are supported by the target system architecture. Node selection is accomplished by repeated random selection with replacement until a 4K region (1 page) is filled. Following this, the nodes are linked, by randomly selecting $Node^{I}_{A}$, $Node^{I}_{B}$ and $Node^{I}_{C}$ for each $Node^{I}$ in the previous selection. This process creates a node network depicted in FIGURE 3. Not all selected node

links are accepted. The network is built under a restriction, which ensures that all circuits existing in the network contain at least one node of each category type. This restriction is essential to ensure that during challenge execution, the calculation does not get into a deadlock, as well as ensuring that all phases of hash calculation and side-effect blending occur, under all circumstances.

A dedicated node, called the ***Prolog*** node, is always the first node to execute. Furthermore, it is executed only once at the beginning. The prolog node is the node that is called to execute the challenge. It is responsible for the initialization of the system in preparation of challenge execution. The prolog configures the hardware side-effect registers, it sets the cache into a known state-0 and configures a dedicated virtual page-table. Prolog node configurations shall be discussed below in greater detail. An additional dedicated node is the ***Epilog*** node, which is the exit node. When the challenge calculation is complete, the epilog node is called to perform some house-keeping chores, restore the system state and then return the calculated challenge result to the caller. The 1st category of nodes, which calculate the hash value by scanning the memory-region, are the nodes responsible for branching to the epilog node once scanning is complete.

#### *3.4.2.2 Virtual Mapping*

A challenge is always accompanied by a virtual mapping, which maps a relatively large virtual address space to a relatively small physical address space. The mapping is determined by randomly selecting a physical page for each virtual address. Naturally, assuming an unbiased distribution, each physical page is mapped to several virtual pages. Page tables used to support a virtual-memory environment normally map a single virtual page to a single physical page [72] within a specific task. Several different virtual pages may point to the same physical page to implement memory sharing among separate tasks. However, in this case, a synthetic page-table needs to be constructed to designate this special mapping.

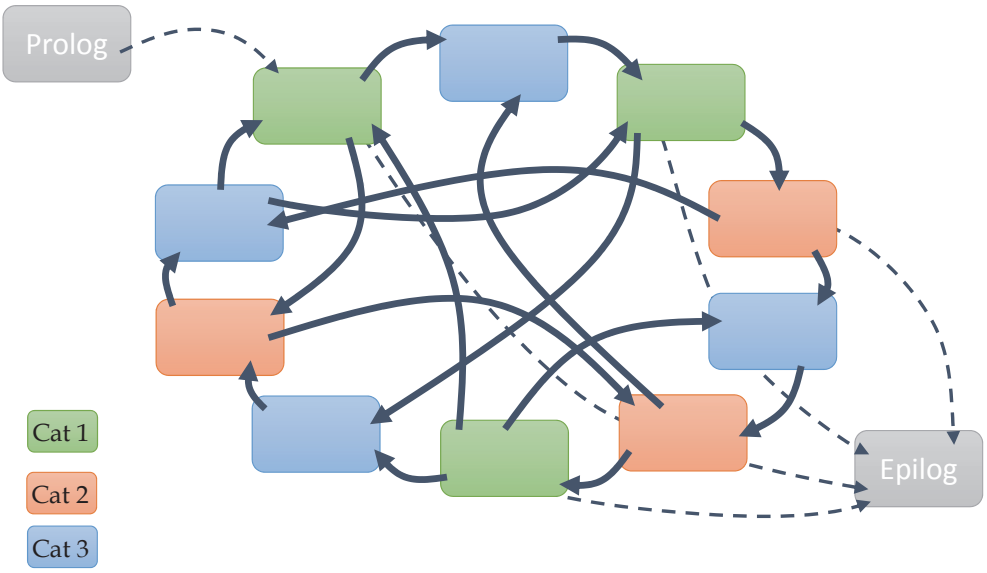


FIGURE 3 Challenge Node Network: Note that every circuit includes at least one node of every category. The prolog node executes first; one of several branches transfers control between nodes, according to the current calculation result; the epilog node completes the calculation and terminates the challenge.

The physical page region shall contain the pages to be attested during the challenge execution. These pages include the virtual-memory page-tables, the page that contains the challenge code and the pages that contain the ***critical components*** of the HDriver. See FIGURE 4. The critical components of the HDriver are a set of routines that, once attested, can be considered as a root for a chain of trust employed to verify the entire contents of the hypervisor. The critical routines include:

(a) Hypervisor initialization
(b) RSA encryption
(c) RSA signature verification
(d) Communication with the external attestation server
(e) Challenge execution.

The main purpose of designating critical routines is to confine the code requiring attestation by challenges to a concise kernel that is not expected to frequently change over time. Thus, future hypervisor versions that are based on a constant kernel of critical-routines can utilize existing challenges.

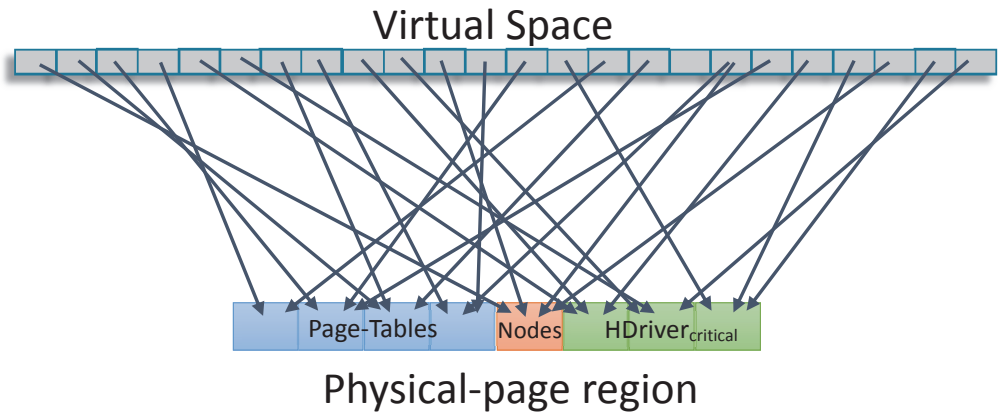


FIGURE 4 Challenge Virtual Mapping. Each physical page is mapped by multiple virtual pages. The Page-Tables are a synthetic construct that reflect the virtual mapping. The Nodes page contains the nodes that comprise the challenge and the HDriver pages contain the HDriver's critical function code, whose contents verification is a major goal of the attestation procedure.

#### *3.4.2.3 Challenge Memory Scan*

The hash calculation order is governed by a pseudo-random-walk (FIGURE 5) according to an LFSR (Linear-Feedback-Shift-Register) generator [73]. Every virtual-space address is visited once, however, as a result of the special virtual mapping, physical addresses are visited multiple times. This is designed to induce side-effects. In a hashing, category 1 node (see: TABLE 1), the value at each visited address is accumulated into the current hash result. The next address to visit is then calculated according to the LFSR function and if it returns to the preliminary address (scan is completed), control is transferred to the epilog (the exit node). Other node types perform additional actions on the current result, such as convoluting the result with a hardware event counter value, but do not advance to the next address location.

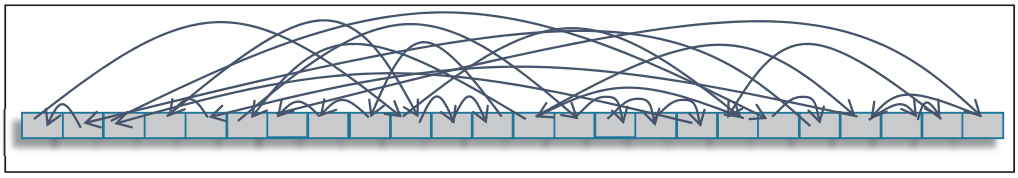

FIGURE 5 Pseudo-random walk to scan the virtual memory-space, using the LFSR algorithm. Each word in the virtual-space is visited exactly one time in a pseudo random order.

The nodes that make up the challenge are located in a single physical page. However, as noted above, this page is mirrored in several virtual pages by virtue of the synthetic virtual mapping described above. The challenge code is constructed so as to execute each node in a separate virtual page. Therefore, control transfers between nodes are designed to take the long-jump, across the virtual address space, rather than the short jump inside the challenge page. This matter of things is depicted in FIGURE 6.

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 convolution type nodes, incorporate a transient hardware counter result into the accumulated hash value. Thereby, both changing the current result value, as well as affecting node progress flow.

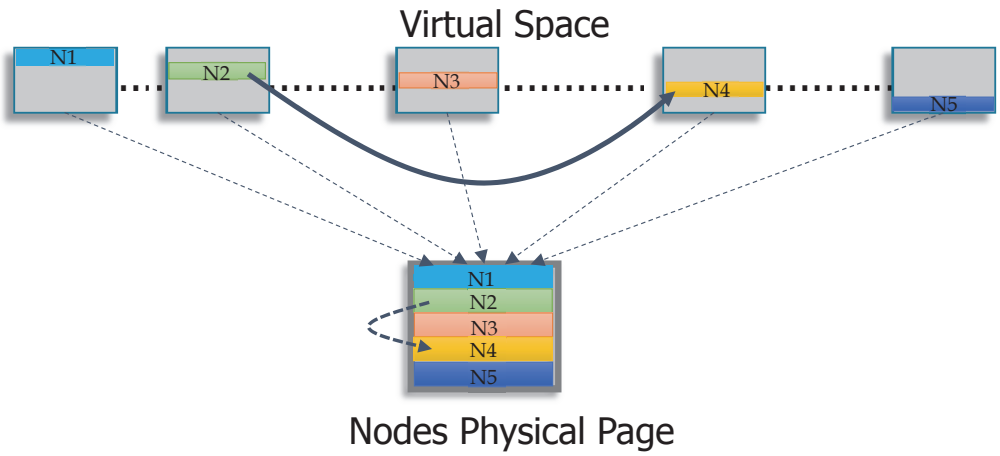


FIGURE 6 Control Transfer between Nodes. All nodes are replicated in all virtual pages that are mapped to the challenge physical page. However, each node executes from a separate virtual page. The figure illustrates a transfer of control from node N2 to node N4 in the Physical and Virtual spaces.

It is stipulated that challenge results calculated in an environment that is different than the intended will generate a significantly different challenge result and thus be easily detected (for example, an attempt to execute the thin-hypervisor under an emulator or as a nested-hypervisor).

The possibility of calculating a correct result by means of emulation shall also be impossible within the allotted timeframe restriction.

### 3.4.3 Challenge Repeatability

An incorrect challenge result (or a result delivered past the time restriction) is considered a means to detect one of the 6 points discussed in section 3.2 above. Conversely, a correct result proves authenticity and trustworthiness of the

attested system – therefore, it follows that the result must be 100% repeatable, given repeated runs on any authentic system.

Challenge repeatability can be tested by running the challenge multiple times on the same system, as well as running it in other systems of the same type and configuration containing the same $\text{HDriver}_{\text{critical}}$ routines.

Furthermore, repeatability allows correct challenge results to be determined empirically, by executing the challenge on a test machine and recording the result. Repeatability is therefore defined as a required property of challenge generation.

To achieve repeatability, the challenge must start by putting the system into a well know state-0. This is the responsibility of the ***Prolog*** node. In addition, all node operations must be deterministic in nature, so that repeated executions, starting at state-0, always produce the same exact result.

To ensure repeatability of a challenge execution, two main prerequisites must be upheld:

(a) The contents of the physical-section (being hashed) must be the same
(b) The values read from hardware-side-effect counters must be deterministic every time they are accessed.

Upholding the former requirement is generally simple. The physical-section contains 3 parts, as displayed in FIGURE 4. The pages of the critical routines in HDriver are designed to remain constant, even between progressing versions of HDriver. The challenge page is identical by definition. The page-tables, which describe the virtual to physical mapping associated with the challenge are kept identical by always using the same predetermined physical addresses for the physical-section. Situations where a particular physical-section may not be available are resolved by allotting for several predetermined options as described later in section 3.5.3. Upholding restriction (b) is much more involved, as will be elaborated in the following paragraphs.

***Prefetch Optimization***: recent generations of modern processors have seen great advancement in pipeline optimizations, to gain significant improvements in throughput. These include branch-prediction circuits and pre-fetcher units. Statistically these predictive actions have a positive effect on performance – effectively increasing overall throughput. However, side-effect event counters are affected as well, leading to seemingly non-deterministic count results. For example, consider counting L1 data-cache hits: when a load operation causes a new cache-line to be filled it is normally not counted as a hit. However, if that cache-line happened to be previously pre-fetched – the load operation will be counted as a hit. Such may be the case with Intel's Instruction pointer based stride-prefetcher [74]. In the event that consecutive memory-read operations with equal offsets (strides) the following access will cause a prefetch of the next cache line. Since prefetching is reliant on several system-load conditions, it does not always happen – leading to a non-deterministic hardware-event count. To uphold determinism an anti-stride-prefetch

algorithm must be used, implemented by intermittently accessing words on even and odd address boundaries when accessing memory to calculate the hash. The pseudo-code to achieve this can be described as:

```
offset ← 0
even_address ← start_even_address_even
LOOP
    access address [even_address+offset]
    increment even_address by step_even
    offset ← offset XOR 1
ENDLOOP
```

***Out of Order Execution***: Modern processors optimize execution throughput by adding support for out of order execution of the instruction stream [75]. Instead of the processor remaining idle while waiting for the current instruction to complete execution, the processor is allowed to continue fetching and processing the next instruction only if it is independent of the results of the current instruction. A synchronizing mechanism exists to verify the allowable conditions and put all the results in order. In Intel processors, for example, this will also effect the L1 data cache [76]. Due to out of order execution, situations exist where reading the side-effect counters associated with cache hits or misses may be non-deterministic. To uphold determinism, the instruction pipeline must be brought up to date ("synchronized") before accessing the hardware-event counters. In Intel processors, for example, this is achieved with the LFENCE or MFENCE instructions.

***Cache Eviction Policy***: Cache eviction policy has a significant effect on processor throughput optimization. This issue becomes even more acute but also more complex in multi-processor environments. Processor manufacturers invest great effort in perfecting cache eviction policies, however, for trade reasons these remain mostly undocumented. Traditional cache-eviction policies such as LRU or MRU are deterministic in nature. Given a fully-evicted (i.e. empty) cache architecture and a known stream of memory access operations will generate a deterministic cache contents. Therefore, an additional operation will produce a cache-hit or cache miss repeatedly. Modern processor, however, are employing cache eviction policies that are much more elaborate than these and in which applying a known memory access stream to a fully-evicted cache does not display deterministic results. Several articles study the cache eviction policies in multi-processor environments Last-Level-Cache (LLC) [77] [78] [79].

Intel processor first level cache (L1 cache) is structured as a 32K-byte, 8-way set associative cache. Each cache line contains 64 bytes. Therefore, every 4K-byte page contains 64 possible cache-lines. Two cache-lines at similar offsets in two separate 4K pages will, therefore, share the same L1 cache set. When all 8 sets of a specific cache-line offset are populated, the next cache-line to enter the set shall evict one of the existing cache-lines already stored in that set.

To establish the determinism of the Intel L1 cache eviction, we performed the following trial. After fully evicting the processor cache, 8 reads are performed from offset 0 of 8 sequential 4K pages (labeled 0 to 7). Since the cache was initially empty, the read cache-lines are stored in the 8-ways of the cache-line corresponding to a page's offset 0. Following that, a 9th read is performed from offset 0 of a different page. This read must perform an eviction of one of the previous (0 to 7) cache-lines to make room for the new line. To determine which cache-line was actually evicted, the first 8 pages are re-accessed in the same order as initially. A hardware side-effect counter that monitors L1 cache-hits is monitored after every page access to determine the first page index that did ***not*** generate a hit. Evidentially this is the page that was evicted when accessing the 9th page.

This trial run is performed separately for every processor, while all other processors are kept dormant at a HALT condition, so their activity will not pollute cache contents. The entire trail, for all processors, is repeated several times to collect meaningful statistics. TABLE 2 displays the recorded results performed on an Intel Nehalem i7-2400 8-processor system.

TABLE 2 Indexes of cache-lines selected for eviction when accessing a 9th cache-line after filling all 8-ways of the cache set

| | **Trial** | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
| **Proc 0** | 6 | 1 | 0 | 1 | 6 | 1 | 0 | 1 | 6 | 1 | 0 | 1 | 6 | 1 | 0 | 1 |
| **Proc 1** | 5 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| **Proc 2** | 5 | 1 | 4 | 1 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| **Proc 3** | 0 | 6 | 1 | 0 | 1 | 6 | 1 | 0 | 1 | 6 | 1 | 0 | 1 | 6 | 1 | 0 |
| **Proc 4** | 7 | 5 | 7 | 5 | 7 | 5 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| **Proc 5** | 0 | 1 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| **Proc 6** | 6 | 1 | 0 | 1 | 6 | 1 | 0 | 1 | 6 | 1 | 0 | 1 | 6 | 1 | 0 | 1 |
| **Proc 7** | 5 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

It can clearly be seen from TABLE 2 that the evicted page is not selected consistently, given the same start conditions and the same memory access stream. Clearly this will obstruct generating a repeatable challenge calculation, since the cache patterns will not be consistent during the memory hashing procedure. Furthermore, note that most of the evictions were the last read page (index 7) pointing to a preference towards MRU.

The foregone conclusion of these trials is that there are additional criteria buried in the cache logic that govern its preferred eviction selection. There are additional well defined cache eviction policies that may be in use in Intel's L1 cache eviction engine. However, there is no formal documentation available. Rather than reverse engineer a specific cache eviction policy, we opted for a more generalized solution to this problem in the form of a "***cache training***" approach. According to the hypothesis that the cache contains an internal finite-state-machine that determines the best eviction policy and assuming that this

state-machine cannot be manipulated by conventional software access, it was contemplated that pre-training the cache could instigate a well-defined and repeatable state that would lead to consistent cache eviction. To prove this hypothesis, the same cache eviction trials were reapplied, however, before each trial commenced the cache was trained by repeatedly applying the memory access stream (i.e., reading offset 0 of 9 consecutive pages). The following pseudo code summarizes this test trial:

```
invalidate cache                                    //Empty entire cache
do 2-times: {read offset-0 of 9 consecutive 4K pages}  //Train cache
read offset-0 of 8 consecutive 4K pages             //Fill 8-way set
read offset-0 of page 9                             //Cause eviction
for pg=0, 1, 2 until 7
   read offset-0 of consecutive page pg
   if (NOT cache-hit)
      return  pg                                    //pg was the evicted
```

TABLE 3 depicts the results of this modified trial run. It clearly shows that cache-training unmistakably generates consistently repeatable results. Note that once the cache is trained the preferred eviction policy is LRU, since the least-recently accessed page (index 0) was evicted as a result of accessing the 9th page.

TABLE 3 Indexes of cache-lines selected for eviction when accessing a 9th cache-line after filling all 8-ways of the cache set with a preliminary cache-training procedure

| | **Trial** | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
| **Proc 0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Proc 1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Proc 2** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Proc 3** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Proc 4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Proc 5** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Proc 6** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Proc 7** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | | | | | | | | | | |

To re-verify this conclusion an additional try was attempted. This time the first 8 pages were accessed in the following random order: {3, 0, 5, 2, 1, 4, 7, 6}, both during cache-training and page access. The detection procedure after reading the 9th page was still conducted sequentially from 0 to 7.

TABLE 4 depicts the results and shows that page index 3 was invariably evicted, supporting the notion of LRU on a trained-cache.

TABLE 4 Indexes of cache-lines selected for eviction when accessing a 9th cache-line after filling all 8-ways of the cache set according to a random sequence and with a preliminary cache-training procedure.

| | **Trial** | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
| **Proc 0** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| **Proc 1** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| **Proc 2** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| **Proc 3** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| **Proc 4** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| **Proc 5** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| **Proc 6** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| **Proc 7** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

To summarize, challenge repeatability must be upheld by the individual node types that comprise the challenge.

The Prolog node, which conditions the system execution has the responsibility of invalidating and then training the cache. To do so it applies the exact memory access stream that will later be in effect when calculating the hash of the virtual memory region. This is achieved by generating the same pseudo-random walk governed by the LFSR algorithm described in section 3.4.2.3.

The hash-calculation nodes, which access memory, must use the anti-prefetcher procedure. This is achieved by dedicating a CPU register as the access offset and repeatedly XORing it with 1.

The hardware side-effect counter blending nodes, need to execute synchronization instructions to defeat the out-of-order execution engine before accessing hardware side-effect counters.

## 3.5 Attestation Flow

### 3.5.1 Overview

The attestation process begins when the thin-hypervisor needs to be established on the target system. This occurs as part of the operating system boot-up procedure. After completing preliminary boot-up chores, the OS brings up registered drivers. When the HDriver is instantiated it attempts to install the thin-hypervisor and perform an attestation procedure to establish a trusted mode of operation.

The attestation procedure entails a ***four-way*** communication handshake with an external attestation server, depicted in FIGURE 7. The target system initiates communications by connecting to the attestation server and sending out a record that includes identification information. In response, the attestation

server sends the target a challenge and its associated virtual mapping. It also starts a time measurement. Upon receiving the challenge the target system sets up a procedure to atomically execute the challenge and install the thin-hypervisor. This operation is performed in parallel on all cores of a multiprocessor system. Once complete, the target system has the thin-hypervisor installed on all of its cores, as well as challenge results (one result for each processor). Challenge results are sent to the attestation server, which evaluates them by verifying that they are both correct and delivered within a pre-allotted timeframe. The attestation server also verifies that the same (correct) result was delivered by ***all*** processors.

### 3.5.2 Hypervisor Initialization

In a multiprocessor system, each processor contains the provisions to support its own separate hypervisor. Different hypervisors may be installed on each processor. However, in the case of a hypervisor, which is designed to protect a system, each processor must be installed with the same exact copy. Hypervisor initialization, on all processors, needs to occur between receiving the challenge and virtual-mapping from the attestation server and before replying with the challenge-result and random material. Once the system is ***virtualized***, i.e., the hypervisor is in control of the system, the hypervisor can protect itself from subversion. However, during the period of its upbringing and initialization it is in a vulnerable state, where malicious code may have the opportunity to make changes to the hypervisor code or data structures in an effort to create ***backdoors*** [80] [81] through which the system can later be subverted.

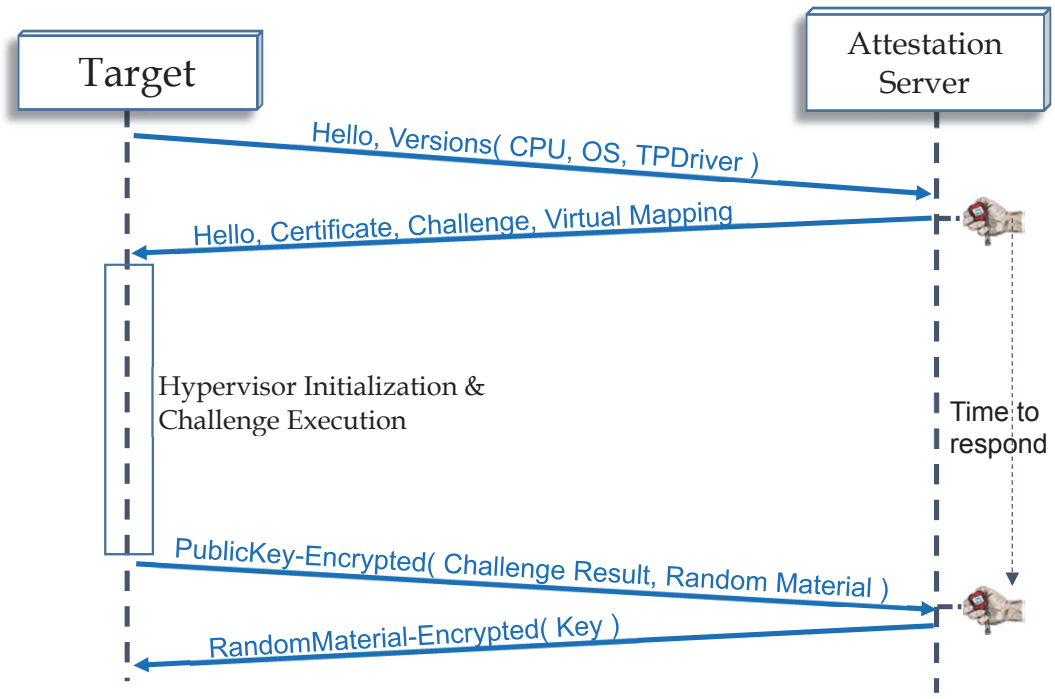


FIGURE 7 Four-Way Handshake: Attestation-Server ←→ Target. (I) Target identifies itself and defines its hardware and software platform parameters; (II) Server administers a challenge + Virtual-mapping. It may also identify itself with a certificate; (III) Target responds with challenge result and random material, encrypted with the server's public-key; (IV) If the challenge result checks-out and was replied within the time constraint, the server replies with the secret-key, encrypted with the random material sent to it in packet (III).

To mitigate this predicament a critical section is applied, into which all the processors in the system must simultaneously enter, execute the challenge provided by the attestation server and instantiate the hypervisor. While each processor is executing the challenge, all other processors must be dormant to ensure they are not executing malicious code. Furthermore, all processors must be at the highest IRQ level [82] with all interrupts disabled to ensure that only the challenge code executes on a single processor while all other processors remain dormant. Once in the critical section, all instructions executed must be verifiable by the attestation result to ensure that the initialization procedure has not been maliciously tampered with. This implicates, for example, that OS routine calls cannot be made.

Before entering the critical section, a portion of memory is allocated from the system's non-page-pool [83]. This memory area shall subsequently accommodate the data-structures required to support the hypervisor on all processors. The reason for allocating memory from the kernel-mode non-paged-

pool heap is to ensure that future access to any of these pages does not cause a page-fault (which can cause a system fault if occurring at a high IRQL). Since at this stage the critical section has not yet begun, the memory allocation is performed by a system call to the OS.

Memory allocation is performed by an arbitrary processor that is assigned the task of bringing up the hypervisor. After allocating memory, the processor issues an IPI generic call to schedule the function containing the critical section process, on all other processors. Since an IPI (Inter Process Interrupt) is mapped to a high IRQ level (14), only one level below the highest level, which is used only for halting the system as a result of a BugCheck, all processors in a running system must immediately respond to an IPI [82].

In response to the IPI scheduling, all processors enter the critical section, where the IRQ level is raised to the maximum value and all interrupts are disabled. Once inside the critical section, all processors accept the first (normally processor 0) are HALTed. Only a single processor is allowed to continue to execute the challenge code whilst all other processors are kept dormant in a HALT condition.

When the single processor completes the challenge execution, the result is stored in a CPU register and the processor issues an IPI to release the next processor from dormancy and allow it to proceed into challenge execution. The processor then becomes dormant by HALTing in its current position. This process continues until all processors have completed execution of the challenge one by one. The last processor to execute the challenge, releases all other processors to continue through the general process.

All processors generate a true-random number that is stored in a privileged CPU register and then all but the first processor become dormant again. The first processor completes a global-initialization followed by a local (private) initialization and an RSA encryption of the challenge result concatenated with the random number using the attestation server's Public-key. It then releases the next processor and becomes dormant. The next processors repeat this process for their local initializations (since the global initialization needs to be performed only once). The last processor to complete its local initialization, releases all processors from dormancy, who then commence to implement hardware virtualization and enter the (safe) hypervisor mode.

Once virtualized, each processor can safely lower its IRQ level and exit the critical section. At this point each processor contains the random number it generated in a privileged register (protected by the hypervisor) and possesses an RSA-encrypted record containing the challenge result and random number. This record shall be sent to the attestation server. It is stipulated that if the challenge result is correct the hypervisor as well as the initialization process can be trusted. Otherwise it cannot. Judgement of this is the responsibility of the external attestation server and as mentioned above, depends on a correct response produced within a limited timeframe.

The critical section procedure can be summarized using pseudo-code as:

Start Critical Section:

*RAISE_IRQL*

*if (NOT FIRST PROCESSOR) { HLT }*

*DR0 ← challenge()*

*if (NOT LAST) { SEND_IPI(NEXT) & HLT } else { SEND_IPI(ALL) }*

*GENERATE RAND*

*if (NOT FIRST PROCESSOR) { HLT } else { initialize_global() }*

*RSA_ENCRYPT(DR0 | RAND)*

*initialize_local()*

*DR0, DR1 ← RAND*

*if (NOT LAST) { SEND_IPI(NEXT) & HLT } else { SEND_IPI(ALL) }*

*VIRTUALIZE*

*LOWER_IRQL*

End Critical Section:

The diagram, displayed in FIGURE 8, depicts the timeline associated with the hypervisor-initialization critical section execution timeline in a 4-processor environment.

***Global initialization*** is performed once during the hypervisor initialization process. It includes setting a variety of control fields that have bearing on managing the hypervisor's activity. In addition, hypercalls are registered to enable their future use. Hypercall function call-backs are used to allow guest functions to request services from the hypervisor.

***Local initializations*** are performed for every processor. These initializations include setting up the Interrupt-table, the hardware virtualization control structures for the Guest, for the Host and for general Hypervisor control, as detailed below.

IDT tables define the addresses of interrupt service routines. The local initialization sets up a separate IDT for the Host and for the Guest. The Guest IDT is configured to allow the hypervisor to intercept interrupts which are not exceptions, i.e. interrupts whose vector is above 32. The host IDT defines interrupts service routines which record the vectors of the occurred interrupts, thus allowing the hypervisor to inject them to the guest later.

Local initialization also addresses the hypervisor control structure, which contains three parts:

***Guest area***: defines the segment registers and other special purpose registers which will be used by the guest upon its activation. The guest area is mostly

filled with the current values of the corresponding registers. The reason for this is that the thin-hypervisor transposes the original OS and its application-stack to the guest. The original OS continues its operation as a guest, immediately following the initialization procedure, from the same point it launched the hypervisor initialization.

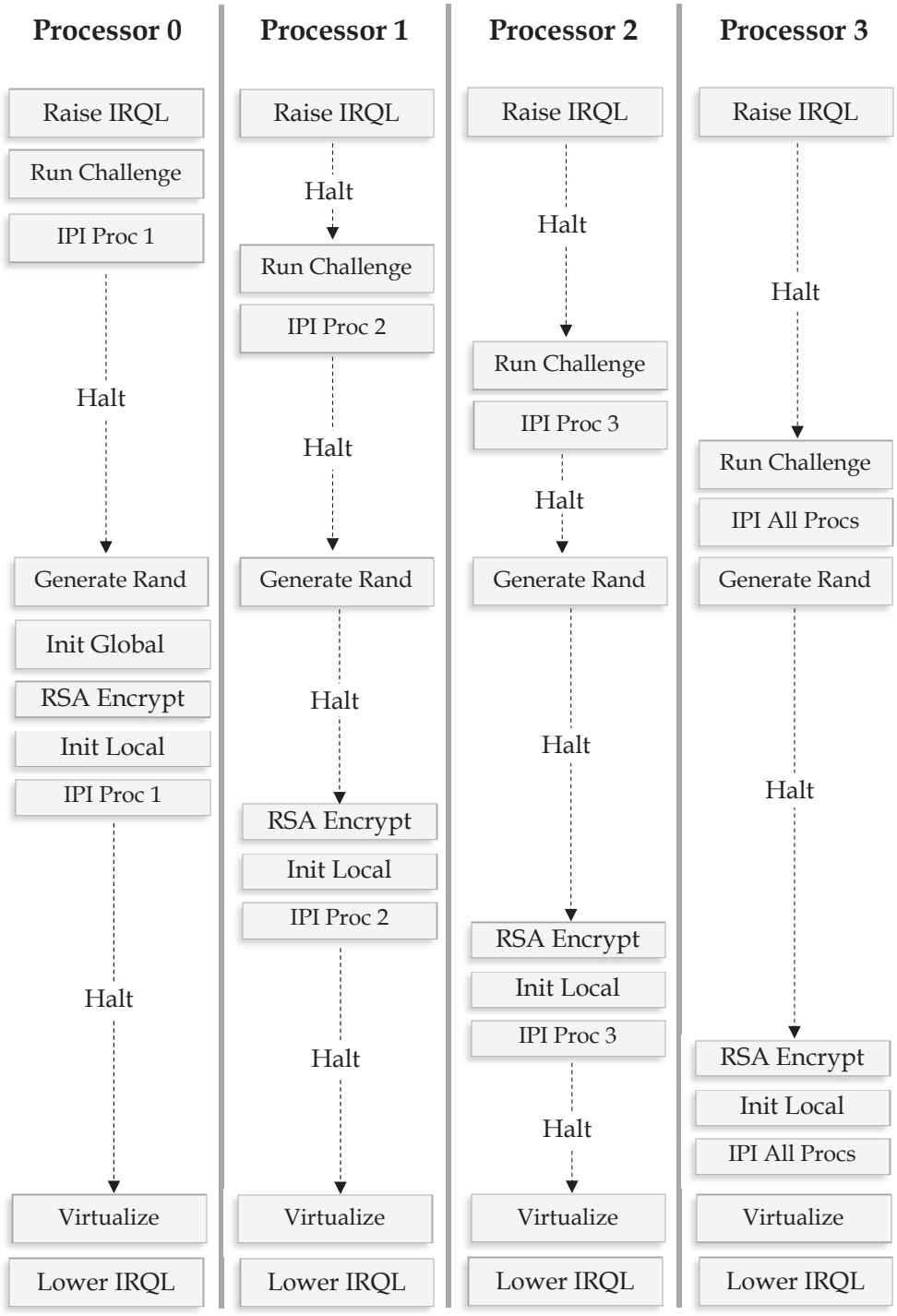


FIGURE 8 Timeline diagram of hypervisor initialization critical-section. This scheme ensures that challenge execution and hypervisor configuration occur one core at a time while all other cores are dormant.

***Host area***: defines the segment registers and other special purpose registers which will be used by the host upon VM Exit. The host area defines the IDT register to point to the host IDT table which was previously allocated and configured. Likewise the CR3 register is defined to point to the hypervisor's pre-allocated and pre-configured page tables hierarchy. The instruction pointer register is set to hypervisor's VM Exit handling routine.

***Control area:*** defines the events that cause VM Exits. In particular this area defines the exception vectors that the hypervisor wishes to intercept. Likewise, this area holds a pointer to the SLAT hierarchy.

Modern hypervisor technology adds a second-level-address-translation (SLAT) [84]. Previously, hypervisors used shadow pages to manage memory separation between Guest virtual-machines running on the same host. The SLAT addresses the associated shadow-page overhead by adding a translation phase in hardware, in addition to the default virtual-to-physical translation occurring routinely in paged-mode operation. The secondary translation occurs only while the processor is in Guest mode and translates the Guest's (virtual) physical address to a real machine (Host) physical address, as depicted in FIGURE 10. Intel implements SLAT as EPT [85], while AMD implements it as RVI [86].

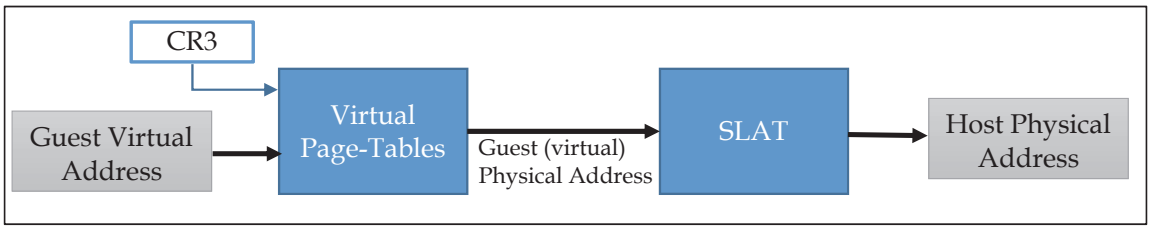


FIGURE 9 Translating Guest virtual address to Host physical address with SLAT

SLAT configuration includes both a mapping of Guest physical address to Host physical address as well as Read/Write/Execute rights that pertain to the Guest. Hypervisor utilizes SLAT as a means to protect the memory regions that contain its vulnerable configuration structures and, as described in the next chapter, to manage and control Guest applications execution rights.

The SLAT is initialized as part of the local initialization performed by each processor. Generally, the SLAT is configured to a 1-to-1 mapping between Guest Physical Addresses and Host Physical Addresses. However, access rights for pages containing the hypervisor code as well as its configuration structures that need to remain out of the Guest's reach, are turned off. Therefore, they appear as non-existent to the Guest. By virtue of this, all hypervisor code and control structures, described above, which contain configurations that oversee hypervisor behavior, among them configurations that render the hypervisor

capable of protecting itself from subversion, are completely out of reach of the Guest.

### 3.5.3 Challenge Execution

Challenge execution occurs on each processor separately while all other processors are dormant in a HALT condition, as detailed in section 3.4.3. Before challenge execution can commence the environment needs to be prepared to support it, as shown in FIGURE 4. This entails:

- Allocating the appropriate physical section in the non-paged pool
- Configuring a page-table that reflects the pseudo-random Virtual-Mapping, see para 3.4.2.2.
- Copying the challenge code contents (nodes), received from the attestation server, into the ***Nodes*** physical-page
- Copying the critical section of the HDriver into the appropriate physical-pages that will be attested by the challenge

Physical page allocation from the non-paged pool needs to start on a prescribed physical address and contain a fixed number of consecutive physical pages. Hence ensuring a deterministic page-table, whose contents must agree with the page-table used when precomputing the current challenge result. The actual allocation is performed by a system-call to the OS. Since a free consecutive-page section in the non-paged pool depends on the previous allocation order and cannot be guaranteed, the actual allocation requested is larger than the required amount. Every challenge result is pre-calculated for several physical-section locations. This allows shifting inside a successful allocation to one of the locations for which a pre-calculated challenge result exists. This is depicted in FIGURE 10. Challenge results are pre-calculated for physical-sections A, B, C and D. The cyan arrows below depict larger allocation requests. Note that the first 7 (depicted) allocation results allow using the A location, while the 8$^{th}$ (last depicted) allocation alternative can use the B location. The same idea is applied for other possible alternative allocations. It is assumed that at least one alternative allocation position shall succeed.

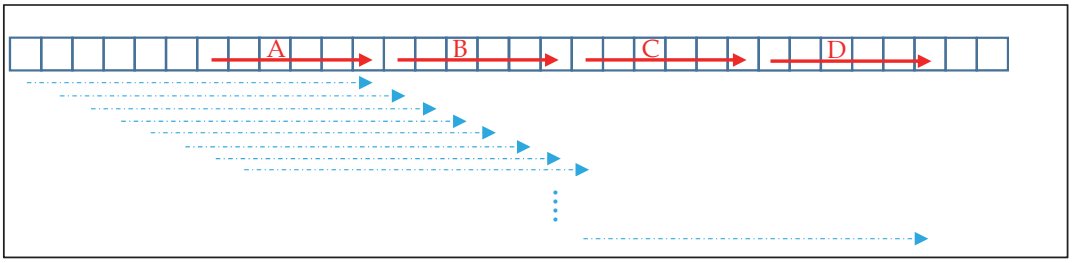


FIGURE 10 Pre-calculated challenge results at locations A, B, C and D. During runtime a larger allocation is requested, thus for any available allocation position (as described by the cyan arrows) at least one of the pre-calculated challenge results can be used.

Page table configuration includes setting up a virtual page-table that reflects the virtual mapping table received from the attestation server. It always sets up virtual address 0 to point to the challenge page, where the Prolog node is always located at offset 0. Therefore, once this page-table prevails, the Prolog node is located starting at offset 0.

The first physical pages of the allocated physical-section contain the root pages of the page-table, followed by the challenge page and then followed by the critical section of the HDriver. Additional pages required to reflect the virtual mapping are used after the HDriver pages. See FIGURE 11.

Page table formats are governed by one of three possible processor modes [87]:

- 32 bits non-PAE
- 32 bits PAE
- 64 bits

After allocating and configuring the page-table the challenge nodes are copied to the appropriate page (physical page 2, 3 or 4 in the allocated physical section). Similarly, the pages containing the critical routines of the HDriver are copied to the pages following the challenge page. At this point the challenge can be executed by transferring control to the prolog node at offset 0 of the challenge page. The prolog node completes the initialization requirements before the challenge nodes can be executed. These initializations include pointing the processor's CR3 register to the page-table root page, thereby enforcing the virtual-mapping, invalidating all processor caches and configuring the hardware side-effect counters.

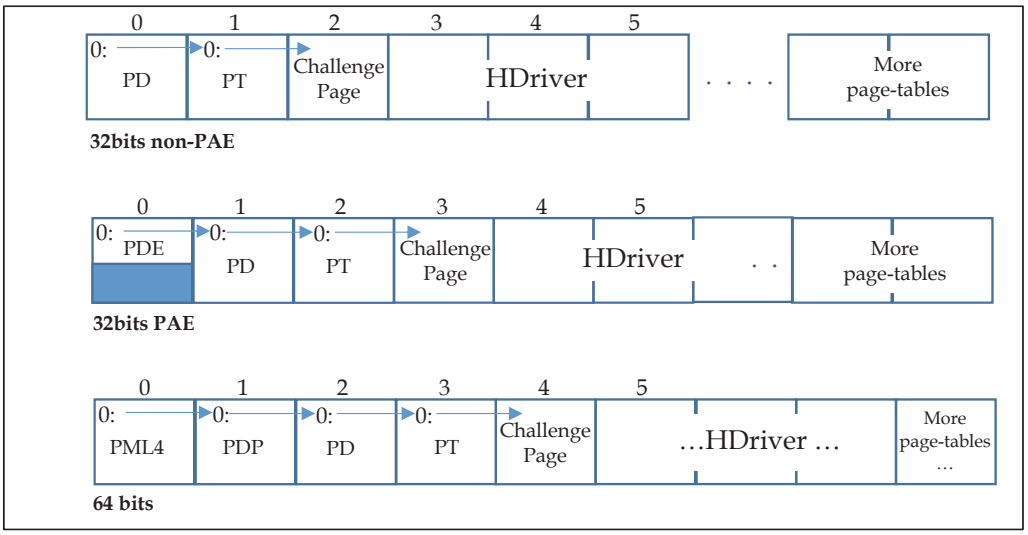


FIGURE 11 Three modes of challenge page-tables.

### 3.5.4 Secondary Attestation

The HDriver code contains the routines required to communicate with the attestation server, initialize the hypervisor, setup challenge execution, as well as the hypervisor itself. These routines are subdivided into 2 sections: section A and section B. Section A is considered critical, since it contains the routines required to perform the hypervisor initialization and the local attestation of section B, based on RSA signatures. Section A is the critical section copied to the attested physical section. Therefore, the routines in section A are the ones attested by the challenge. Furthermore, they are not expected to change in the future, consequently challenge results shall remain intact even if the HDriver evolves, so long as changes are confined to section B.

By performing a challenge attestation of section A, we can trust section A (assuming the challenge result is correct) to perform a trusted attestation of section B based on RSA signatures. This architecture constitutes a chain of trust [88].

## 3.6 Secure communications

The four-way handshake communication protocol between an attestation server and a target system detailed in FIGURE 7 must be performed securely. Thereby ensuring that the secret information eventually transferred to the target system, should it pass the attestation challenge, will not fall prey to a malicious assailant. Since the communication flows between two remote systems, no assumption is made regarding the safety of the data communicated. It is assumed that all communications may be subject to recording. Moreover, since the assailant may have ownership and physical access to the target system, it is further assumed that communications may be recorded after arriving at the target system.

Detailed Security measures for each type of transmission:

1. Target ➔ Auth. Server: In this preliminary transmission the target identifies itself. None of the information communicated requires concealment, therefore the transmission occurs openly.
2. Auth. Server ➔ Target: The attestation server reply to the target contains the challenge-code, the virtual mapping and possibly the server's certificate. Since challenge and virtual mappings are transmitted once only and never repeated, there is no need to conceal their contents. Similarly, a server certificate contains only public information and therefore does no need concealment either.
3. Target ➔ Auth. Server: In the course of this transmission, the target transfers the challenge execution results (one per each processor) to the Server. The target also transfers a random value that is generated by

the hypervisor, which was instated in the target immediately after executing the challenge. It is desirable to conceal the challenge results in order to eliminate the possibility of replay attacks from other systems that may receive this exact challenge at some later time. Furthermore, as will be explained later, the random value generated by the hypervisor, must absolutely remain concealed. To achieve these concealments, the target encrypts the entire reply with the attestation server's public RSA key, which is hard-coded in the HDriver on the target. The challenge results and random value shall also remain concealed from the target OS, since these values are known only to the instated hypervisor, which keeps them concealed from the Guest OS.

4. Auth. Server ➔ Target: Assuming the challenge results are correct, the attestation server will transfer the secret information back to the target system. Naturally, this reply must be concealed. This is achieved by encrypting the entire reply record with the random value received in the 3rd transmission, as key. Since this value is known only to the target system hypervisor, it is the only entity that can decrypt and interpret the secret information. It should be noted that at this point the attestation server can ***trust*** the remote hypervisor, since the challenge results were correct and timely.

To sum, communications for information and secret key material transfers, between the attestation server and the target system hypervisor, during the remote attestation procedure, can be achieved in a fully concealed and secure manner.

## 3.7 Verifying the Attestation Goals

In section 3.2 six attestation goals were detailed. We review each requirement to verify that the attestation methodology described above upholds all the requirements.

1. *Verify that the hypervisor is a root-hypervisor that has control of the machine, i.e., no emulator or primordial hypervisor already exists in the system:*
   The challenge code includes nodes whose function is defined as hardware side-effect inducing. One type of such a node executes an instruction that forces a hypervisor intercept. For example, the CPUID instruction always generates a VM_EXIT. Furthermore, a hypervisor cannot suppress such a VM_EXIT. At best it can immediately return to the Guest. However, a hypervisor intercept must cause a CPU control change to the hypervisor's intercept handler, which in turn causes several cache pattern changes. The instruction cache gets a new cache-line (of the hypervisor handler) and the TLB cache stores the address

translation of the handler address. Therefore, challenge results will significantly deviate from the expected results in a system with no hypervisor, as compared to a system with a Hypervisor.

The only alternative to circumvent this, is if the target system contains an emulator that can track hardware side-effects, such as cache patterns. However, thousands of instructions would be necessary to fully emulate the effects of a few single native instructions. Therefore, an emulated result could not be produced in the predetermined timeframe.

2. *Validate the hypervisor contents*
   Hypervisor contents are inherently verified since the challenge implements a hash calculation of the critical functions, which in turn are used later to verify signatures of the entire hypervisor code.

3. *Verify that the hypervisor that is being attested is the one executing*
   Countering this requirement implies that an adversary has made changes in the challenge code to his advantage but is hashing the original (unchanged) challenge contents in an effort to keep the calculation result correct. By doing so, an adversary may hope to alter the flow of setting up a trusted hypervisor, while still generating the correct challenge result.

   However, making such a change will be detected by the challenge procedure described above, since cache patterns shall develop differently when executing code that calculates a hash of itself as opposed to hashing a different memory location. The reason for this is that when an instruction is executed, the cache line containing that instruction is invariably stored in cache and the address translation of the code's virtual page is stored in TLB cache. Therefore, if the instruction loads a value located within its cache line, a cache hit is guaranteed. However, if that instruction loads a value from a different location, a cache hit is not guaranteed. Actually, when this situation occurs for the first time a cache miss will surely occur. This difference will lead to different challenge results under each condition.

   Alternatively, an adversary may try to make changes to the critical HDriver routines, but use the original HDriver contents in the hashing process, with the intent to later execute the modified code. This situation is restricted, since it is up to the challenge code to determine that the code that will be executed is the one that was hashed. Nevertheless, the challenge code cannot be altered without

detection, as explained above. Therefore, this alternative cannot be realized as well.

4. *Ensure that all cores/logical-processors are running the attested hypervisor*
In multiprocessor systems each individual processor must be attested. To ensure this requirement, the HDriver routine, responsible for execution of the challenge nodes, creates a critical section. An IPI directs all processors to that section, in which each processor, individually and atomically, executes the challenge code and achieves its own challenge result, followed by instantiating the hypervisor. During the execution of the critical section, interrupts are turned off and the IRQL of all processors is set to maximum, ensuring that interrupts cannot occur.

   The outcome of the critical section is that each processor is in Host mode with an enabled hypervisor and each processor contains its individual challenge result. An encrypted record containing a series of challenge results, calculated by all the processors, is then sent to the attestation server.

   On the attestation server's end, two major validations are performed:
   (a) The number of processors expected from the system type identified when communications were setup, is equal to the number of challenge results received
   (b) All challenge results are equally correct and received within the restricted time frame.

   It follows from both these validations that all processors in the target are executing a trustworthy hypervisor.

5. *Create a trusted communication channel to the hypervisor*
Section 3.6 details the provisions taken to uphold this restriction. Confidential information is sent from the target to the attestation server using RSA encryption with the attestation server's public-key. Therefore, it can be interpreted correctly only by the attestation server.

   Confidential information returned from the attestation server to the target uses encryption, based on random material received from the target. An adversary cannot apprehend this information over the communication channel, since it is sent encrypted with PKI. Similarly, it cannot be seized in the target system itself, because it is generated internally by a hypervisor setup to protect its resources.

6. *Transfer a secret to the trusted hypervisor*
   The attestation server will transfer a secret to all the processors of a successfully attested target. As described in the previous point, the secret data is encrypted with random material provided by each of the hypervisors instantiated on each individual processor. Upon receipt, each hypervisor shall decrypt and safely store the secret in a privileged register, which it can protect, since any access to the privileged register is intercepted by the hypervisor. The attestation server can fully trust the hypervisors to protect the secret, since all hypervisors are trustworthy by virtue of point (4) above.

# 4 EXECUTION PROTECTION OF NATIVE CODE

## 4.1 Overview of the methodology

Many operating systems, among them x86 architecture operating systems, enforce control over memory access rights for applications through the virtual paging mechanism. Access rights normally dictate *Read, Write* and *Execute* rights. Virtualization extensions, which were introduced to the x86 architecture by Intel and AMD, allow a hypervisor to control memory access rights at an additional level, below operating systems, through a mechanism called Second Level Address Translation (SLAT). Intel and AMD refer to this mechanism as Extended Page Table (EPT) in chapter 28 of [44] and Rapid Virtualization Indexing (RVI) [61], 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, see chapter 4 [44]). When SLAT is enabled, the memory access restrictions of both the Virtual paging and the SLAT apply in tandem. For example, if Virtual paging allows Read and Write access for a certain page, however SLAT allows only Read access – the net access rights for this page shall be Read-only. 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 access rights for the entire memory and in all memory modes.

The methodology described here, is based on utilizing the hypervisor's control over SLAT to prevent execution of unauthorized software. To protect a system against execution of unauthorized (or arbitrary) native code, the system is first scanned when it is in a known good state. This can take place, for example, immediately after a new system is installed from an original, certified, source – such as a DVD provided directly from the OS manufacturer. The scanner locates all executable files, including .exe, .dll and .sys (driver) files and creates a database of signatures for each code-page (4K granularity) of each executable.

Initially, when the hypervisor starts executing, it configures the SLAT to deactivate the *Execute* rights of all pages, thus effectively receiving an intercept for any execution attempt of the guest. After returning control to the guest,

upon such an intercept, the hypervisor has the opportunity to verify the executing page authenticity, by hashing the page content and comparing it to the precomputed value prepared during the scanning phase. After authenticity is established, the hypervisor grants the page's *Execute* rights but forfeits its *Write* rights, thus setting up to be intercepted in case of any attempt to modify the authenticated *Execute* page. In the event that such an interception occurs, as a result of a modification attempt, the hypervisor grants the page *Write* rights but forfeits its *Execute* rights. Therefore, at all times, a page can have either *Execute* rights or *Write* rights, but never both at the same time.

Code-pages are authenticated by comparing hash value calculated during execution attempts against pre-calculated signatures organized by the scanning program in a database that is kept locally on the target system. After being compiled, the database is signed, in order to prevent any illicit modifications. If additional software needs to be installed in the system, the database of code-page signatures must be extended to accommodate the signatures of the new software's executable files. Since the database must be signed, only an authority capable of signing the modified database can make this addition. The hypervisor can verify the authenticity of the database signature, since it possesses the attestation authority's Public key. Finally, the hypervisor can be trusted to apply the correct key, since it is part of the contents being authenticated by the attestation procedure. Section 4.2 contains a detailed description of the database structure.

## 4.2 Whitelisting an Execution Environment

Application whitelisting, based on pre-determining the list of software application allowed to execute on a system is a well-known methodology for maintaining system security. Windows enterprise versions have built-in support [89], as well as several commercial products on the market that exercise this methodology [90] [91]. However, these products all maintain security at the application-level granularity. Moreover, they are reliant upon OS services to enforce their security. This leaves the door wide open for malicious hackers to exploit 0 days and OS vulnerabilities that allow them to inject executable code, through which persistency may then be achieved. It may suffice to only infiltrate memory with a few single instruction codes to ultimately gain access and control over a computer system. The methodology described herein addresses and amends these deficiencies by maintaining a whitelist at the physical-memory level rather than the application-level, as well as relying upon a trusted hypervisor to carry-out the security enforcement rather than the OS.

We begin our detailed explanation of the rogue-execution prevention mechanism, by describing the structure of the whitelist database that contains the hash values of the code-pages in all the system's executable modules (see

FIGURE 12). This database consists of a collection of descriptors for all executable modules. Each module descriptor contains information for a specific executable (PE file in Windows [92] or ELF file in Linux [93]), 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 locate 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 in the executable, and contains the following fields:

- **Record size** — the size of this section descriptor. This field allows to locate the next descriptor.
- **Offset** — the offset of this section from the beginning of the image file.
- **Length** — the size of the section described by this descriptor.
- **Page[i]** — a page descriptor that corresponds to the $i^{th}$ page of the section. The number of pages in the section is calculated as the section size divided by 4096. Since pages must begin on a page-boundary, partial pages may exist at the beginning or the end of the section. Partial pages also have a descriptor. This descriptor contains the page's ***hash*** and the indexes of the first and last relocation indexes pertaining to this page.
- **Number of Relocs** — the total number of relocation descriptors that follow.
- **Reloc[i]** — relocation descriptor, which contains an ***offset*** into the page where an address needs to be adjusted to account for an application relocation in virtual space. The ***type*** field determines the width of the address to relocate.
- **Number of Datums** — the amount of the datum descriptors that follow.
- **Datum[i]** — datum descriptor, which is required to generate a relocation when it crosses a page boundary.

## 4.3 Enforcing Valid Execution of Native Code

### 4.3.1 Initialization

Recall that the HDriver is the kernel-mode device-driver that instantiates the hypervisor along with the mandatory attestation procedure. The HDriver 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 every time the operating system loads an executable to memory and the second callback is invoked every time a process terminates. Both callbacks update a data structure that represents the memory layout of all the processes that are currently active.

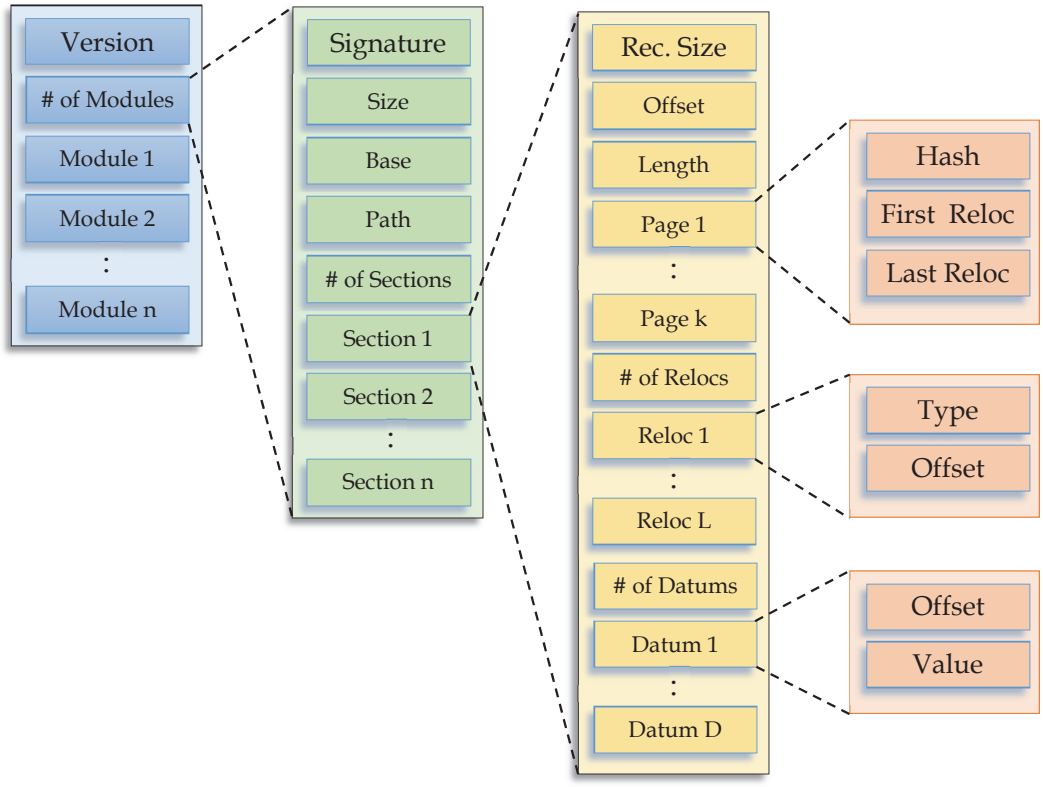


FIGURE 12 Database structure of executable code-page hashes. The database includes a section for each executable module. Each module contains an array of sections and each section contains an array of signed executable pages along with the possible relocation data for each of the pages it contains.

This data structure contains a linked-list of process descriptors. Each process descriptor contains the corresponding ***process identifier*** and a pointer to a linked-list of ***module*** descriptors. Each module descriptor contains the ***load-address*** location in memory of the corresponding module and the whitelist database ***index*** of this module's descriptor. FIGURE 13 depicts this data structure.

The HDriver initializes and installs the hypervisor, which later manages the access rights of physical pages. The hypervisor and the HDriver callbacks must operate concurrently, since the callbacks update the memory layout data structure that is used by the hypervisor. It would be advantageous to guarantee that the HDriver and the hypervisor are the first executables to load in the system. Unfortunately, the driver initialization order is determined by the operating system and cannot be altered. Therefore, the operating system may (and normally does) load and initialize some drivers prior to the HDriver.

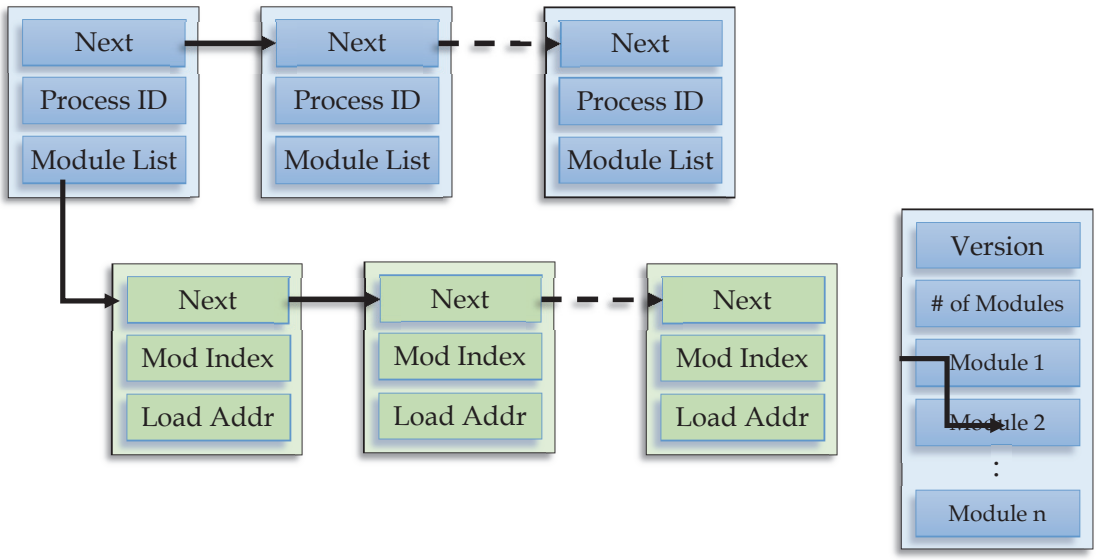


FIGURE 13 Process memory-layout data-structure. The structure is a linked-list of processes currently executing in memory. Each process contains a linked-list of all the modules executing within that process.

Consequently, the callback, which is installed only during HDriver initialization, will not be called for those drivers. The HDriver solves this problem, by traversing operating system-specific data structures that contain information on the drivers that were already loaded. By doing this, the HDriver has the opportunity to properly initialize the process memory-layout data-structure with the processes that are already running in the system. FIGURE 14 presents the data structures that are used by a 64-bit version of Windows 8. A driver's initialization function receives a pointer to the driver object that represents the current driver. The ***DriverSection*** field points to a data structure

that represents all loaded drivers. Each such structure is a member in three doubly linked lists: ***LoadOrder***, ***MemoryOrder*** and ***InitOrder***. Each such list is a cyclic enumeration of the loaded driver ordered by: driver's load time, driver's memory location and driver's initialization time, respectively.

### 4.3.2 Access Rights Modification

Once the hypervisor completes the attestation procedure and its initialization process, it initially deactivates the ***Execute*** rights of all memory physical pages in the SLAT. Therefore, after returning control to the guest, any attempt to execute an instruction triggers a "SLAT Violation" (unauthorized access to physical memory) which causes a VM Exit intercept that passes control to the hypervisor.

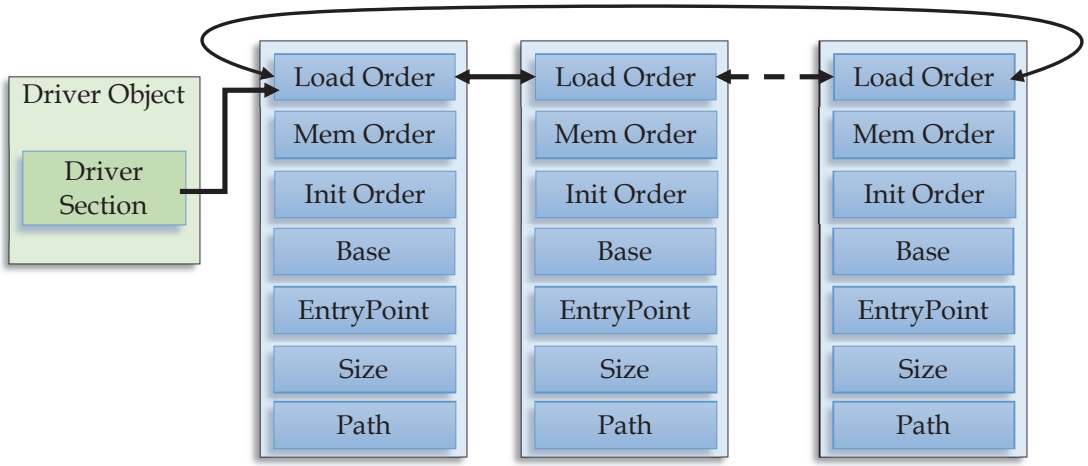


FIGURE 14 Windows 8 linked-list of loaded device-drivers

The hypervisor then has an opportunity to verify the authenticity of the page containing the instruction that invoked the intercept, by validating the page's signature. If the page is not successfully authenticated, the hypervisor signals a breach condition and aborts the offending guest process. Otherwise, if the page is authenticated, the hypervisor will change its access rights to ***Read*** and ***Execute.***

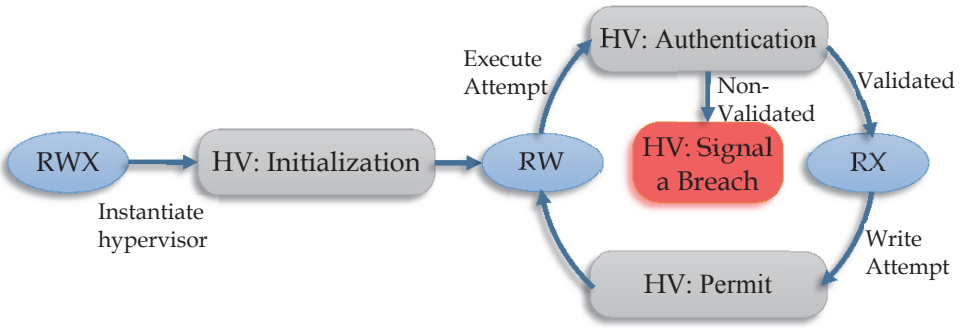


FIGURE 15 Physical page access-rights state diagram. Following hypervisor initialization and attestation, all physical pages have R/W access only. Any such page that is executed will cause a hypervisor VM Exit that will validate the page's signature before allowing it R/X access only. An attempt to write to the page will cause a hypervisor VM Exit allowing it to remove Execute access and restoring the page to the initial R/W access rights.

After returning control to the guest, all instructions in this page can be freely executed without causing additional intercepts. However, any attempt to write to this page triggers a "SLAT violation" and an intercept into the hypervisor, which changes the access rights to ***Read*** and ***Write*** and deactivates its ***Execute*** right. This process is depicted in FIGURE 15. A detailed description of the verification procedure appears below.

In a multiprocessor system each processor has its own hypervisor and a separate configuration structure. In particular, each processor has its own SLAT hierarchy, which can independently (of other processors) specify the access rights for each physical page. The hypervisor must maintain identical configurations of all processor SLAT hierarchies (with a few exceptions, as we will see later) in order to fully prevent execution of unauthorized instructions. To understand how this can occur, consider the following scenario: an authentic page requests 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 for this page, which enable it to modify the contents of the 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 SLAT hierarchies [44]. To solve this problem, whenever the hypervisor on some processor needs to change the access rights of a page, it first sends a request to hypervisors on all other processors to make the intended change in their SLAT. The same change is made on the SLAT hierarchy of the originating processor, only after all the SLAT hierarchies of all other processors were changed.

The inter-hypervisor request mechanism is implemented as follows. During its initialization the hypervisor allocates a constant-size queue of requests for each processor, which represents the outstanding 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. This 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:

(a) It inserts a new element in the requests queue of the remote processor
(b) It sends an IPI (Inter-Process Interrupt) to the remote processor on the special vector (0xFE).

After issuing the request, the hypervisor waits for the changes to be applied. FIGURE 16 depicts the entire process of access rights modification as it is performed in a multiprocessor system. The procedure follows a 7 step sequence: (0) A page access violation occurs in a guest application. The hypervisor on that process intercepts the event; (1) The hypervisor adds access-right change requests to all the ***other*** processor request queues; (2) The hypervisor then sends a special IPI to all other processors; (3) The IPI service routine on all processors generates a hypercall to enter the hypervisor on the receiving processor; (4) each alerted hypervisor fetches all pending requests from its request-queue and (5) performs the required access-rights change in its copy of the SLAT table; the originating hypervisor monitors the other processor's SLAT tables directly and when the requested access-rights change is detected in all other SLAT tables (6) it makes the change in its own SLAT table. This procedure guarantees that all SLAT tables remain in synch after every access-rights change event.

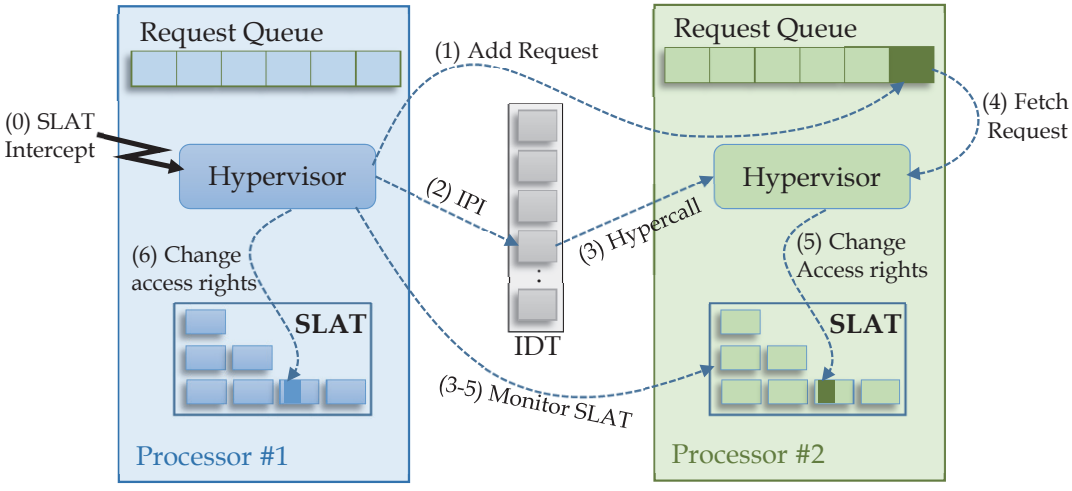


FIGURE 16 Access rights modification in a Multiprocessor environment. Each core has its own SLAT table, therefore changes to the SLAT table in one core must be reflected in the SLAT table of all other cores.

### 4.3.3 Execution Request Verification

The intercept event caused by an attempt to execute an instruction in a memory page that does not have ***Execute*** rights, transfers control to the hypervisor, who then has the opportunity to validate the page. The verification procedure is as a Boolean function returning true only if the verification succeeds. This function has one parameter — the virtual address where the SLAT violation occurred. The verification function performs the following steps:

1) Fetch the current process identifier from OS-specific data structures. FIGURE 17 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 HDriver. 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 SLAT violation. 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 the SLAT (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 SLAT 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 data values
    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 $p_i$ to 0
    c) For each relocation r do:
        i. Hash the bytes [$p_i$..r.offset-1]
        ii. Let d be the datum at offset r.offset; d exists only if the relocation field bytes cross the page boundary.
        iii. If d does not exist, fix the address at r.offset and hash it
        iv. Else, hash d.value and verify that this is the value (relocated) at r.offset
        v. Advance pi to r.offset + length(r.type)
    d) Hash the bytes [$p_i$..{the last byte to be hashed}]
8) Compare the hash result to the expected hash value and return true iff they are equal

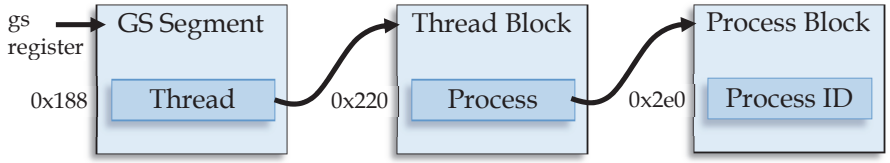


FIGURE 17 Windows 8 Process ID location

The ***Datums*** array holds the values of relocation fields that cross page boundaries. The verification procedure must read the value at the relocation position but, since it operates in kernel-mode, it must not read data that might induce a page fault. This can occur if a relocation field crosses a page boundary into a page that is not currently in main memory. For this reason, values of relocations that cross page boundaries are stored in the special ***Datums*** array. FIGURE 18 presents the most general example of a verification process. Here, only part of the page contains code to be hashed. This code section contains 3 relocated fields, labeled A, B and C. Note that C only partially belongs to the page, as it crosses into the next page. Relocated fields A and B contain modified address values that need to be fixed to obtain their original value. Relocated field C's original value is stored in the Datum array. Therefore, this section will be hashed in 3 iterations:

I. Bytes of 1 and fixed bytes of A
II. Bytes of 2 and fixed bytes of B
III. Bytes of 3 and relevant (included in page) bytes of Datum point C, taken from the Datum array. The verification routine shall also verify that the included bytes of field C contain the expected values after relocation.

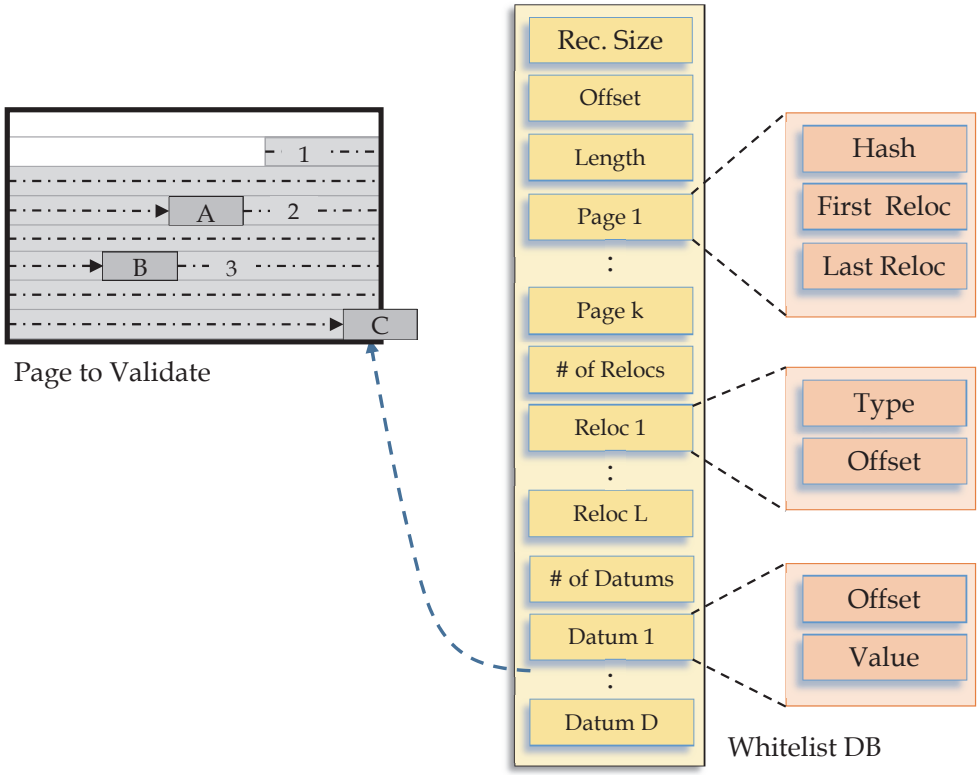


FIGURE 18 Page Validation Process. Relocations need to be accounted for. Their location and width in the section is recorded in the database for this purpose. The intended value is also recorded if the relocation field crosses a page boundary to ensure that bytes past the page are not read at a high IRQL.

## 4.4 Special Execution Pages

### 4.4.1 Mixed Pages

Some pages may contain both code and data. Typically, such pages cross the boundary between a code section and a data section, where those sections are not page-aligned by the linker. The problem these pages pose is that it is unsafe to grant them ***Execute*** rights, since they cannot be authenticated entirely and once the entire page is granted ***Execute*** rights, rogue code could potentially be injected and executed from the un-authenticated data part. However, the code in these pages cannot execute without ***Execute*** rights. The solution to this problem is ***controlled execution***.

In essence, controlled execution monitors execution of individual instructions within a mixed-page, to corroborate that only instructions belonging to the authenticated part of the page are allowed to execute. After

granting the (partial) page ***Execute*** rights, the hypervisor monitors the instruction pointer by intercepting after the execution of every single instruction. During every such intercept, the hypervisor checks whether the instruction pointer has exited the authenticated section, and if so, the hypervisor deactivates the ***Execute*** rights of the page. To enter this mode of operation, the hypervisor sets the ***Monitor Trap Flag***, sections 24.6 and 25.5.2 [87], after granting ***Execute*** rights to a mixed page. After returning control to the guest, this mode will cause a VM exit to the hypervisor on the next instruction following the instruction that is executed upon returning control to the guest (VM entry). The hypervisor maintains this mode of operation until the instruction pointer exits the authenticated part of the mixed page.

Controlled execution is the safest solution to the problem of mixed pages, however, it has dire consequences to execution performance. To alleviate this problem, an alternative approach has been researched and evaluated, albeit not as hermetically secure. According to this alternative approach, modern-processors have the IA32 DEBUGCTL MSR which may be configured to define advanced breakpoint functionality. Coupled with setting the TRAP flag in the processor's EFLAGS register, the hypervisor can configure the processor to generate a breakpoint after every branch instruction, rather than after every instruction. Thus, the hypervisor intercepts all branches that may potentially transfer control outside the authenticated area. When the hypervisor gains control, due to such an intercept, it has the opportunity to deactivate the ***Execute*** rights of the mixed page, if this occurs. Two additional facilities must be established to augment this mode of operation: (a) interrupt activation and (b) fall-off-the-edge. When the former occurs, the processor activates the interrupt service routine without activating single-instruction breakpoints. In this case, the hypervisor must deactivate the mixed page's ***Execute*** rights, since the interrupt service routine may potentially cause execution of rogue instructions in the unauthenticated part of the mixed page. The hypervisor achieves this by taking over the entire IDT table so that any interrupt activation is first captured by the hypervisor, allowing it the opportunity to deactivate the mixed page's ***Execute*** rights. In the latter case, the hypervisor must protect against the potential case of the last instruction in the authenticated part progressing sequentially into the unauthenticated part. Since in this case, control has exited the authenticated part without a branch taking place, an intercept to the hypervisor does not occur. To resolve this difficulty, the hypervisor can make use of the processor's DEBUG registers, which allow setting up hardware breakpoints on a specific addresses. The hypervisor can thus setup a hardware breakpoint on the first address of the unauthenticated part, giving it the opportunity to gain control in this case and properly deactivating ***Execute*** rights. Naturally, this case will also trigger a breach condition. As mentioned above, this performance-improvement comes at the price of hermetic security. This follows from the fact that the hypervisor cannot secure the TRAP bit in the processor's EFLAGS register, as there is no way to

configure a hypervisor intercept when this flag is changed. Exploiting this vulnerability may be difficult but not impossible.

If absolute security is opted, single-step controlled execution based solely on the Monitor Trap Flag must be preferred. Fortunately, the occurrences of these cases are rare and from empirical study overall performance is not significantly degraded.

### 4.4.2 Page Modifying Instructions

A rare, but nevertheless existing type of instruction, writes data to the page from-which it is executing. Such an instruction cannot successfully run in a protected system, while still respecting mutual exclusivity between ***Write*** and ***Execute*** permissions. The hypervisor detects these instructions and solves the conflict by setting the page to ***Write*** permission and executing the special instruction inside the hypervisor by interpretation.

### 4.4.3 Code Pages that Include Data-Sections

Some executable applications contain code-pages that belong wholly to the code section of the executable file, however contain embedded data sections, which the application writes to. The problem with these sections is that these pages and the locations of the unusual embedded data-sections are not recorded anywhere in the executable file. Therefore, they can only be detected and mapped empirically. Fortunately, such case have only been seen in a few OS executable files. The solution to this problem is to augment the scanning utility and to extend the whitelist data-base to allow locating and labeling these pages as special pages and to include the embedded data section addresses and sizes. The hypervisor then treats these pages as mixed pages with embedded data sections, in a similar fashion to that described for mixed pages detailed in section 4.4.1. The only difference being that this extension allots for **several** code and data sections in a mixed page.

### 4.4.4 Self-Modifying Code

Self-modifying code is code that writes out its own instructions as they are being executed. This type of behavior is practiced by Microsoft's "***Patch-Guard***", otherwise known as Kernel-Patch-Protection [94] [95], which is a stealthy part of the Windows OS responsible to protect Windows own integrity. The problem presented by this is similar to that described in section 4.4.2 above with respect to data being written to the same page as the instructions that write them, however it is more severe, since the data being written are instructions that are also executed.

A general solution to this problem does not yet exist. However, from extensive tests run on several versions of Linux and Windows this problem manifests itself only in well-defined cases, such as Microsoft's ***Patch-Guard***. The proposed solution, applicable to well-known discrete instances, is for the

hypervisor to detect and verify that this is the case and respond with: (a) allow a deviation from the strict permissions scheme by granting these special pages full permission rights or (b) quietly disabling its execution altogether and continuing normal execution without raising an error condition. The former solution is unsafe, since it exposes a vulnerability according to which an adversary may devise injected code to masquerade as a well-known implementation, such as Patch-Guard, but eventually generate malicious code. The latter solution has the disadvantage of skipping the execution of code that may be important in the system. However, in the specific case of Microsoft's Patch-Guard, we argue that Patch-Guard is redundant to the protection provided by the hypervisor.

## 4.5 Performance

### 4.5.1 Hypervisor Overhead

When a hypervisor is activated on a system, two major parasitic performance overheads apply: (a) servicing VM exit intercepts and (b) secondary level address translation (SLAT) [96]. The former occurs on every event for which the hypervisor is intercepts the guest and performs a VM exit. Several such events are mandatory VM exits, as detailed in 25.1.2 [44]. Additional events shall cause a VM exit only if configured to do so in the hypervisor's control structures. During a VM exit the processor must record the reason for the exit as well as record the state of the guest (save its control registers) and then load the host state (restore its control registers). This activity is reversed upon returning to the guest when preforming a VM entry. The overhead of these procedures is associated with every intercept and has been measured empirically on an Intel Core i7-3687U to be in the range of between **1194** to **3042** cycles, depending on whether the required host configurations is already in cache or not, respectively. Naturally, an exact value depends on the processor. To calculate the total associated overhead, the number of cycles spent inside the intercept service routine needs to be added and is, of course, a function of the tasks to be performed in each intercept.

This type of performance overhead is minimized in a thin-hypervisor, as compared to a full-virtualization hypervisor used to manage several concurrent VMs, since only a handful of intercepts, needed to protect a single VM (and the hypervisor itself), are configured to VM exit.

The latter performance overhead source (b) is inherent to activating the secondary level address translation (SLAT). In a system with a hypervisor that activates SLAT, every memory access that misses the TLB is associated with additional CPU cycles that consult the SLAT to perform the necessary translation from guest-physical to actual-physical memory address as well as verifying access rights. The additional cycles accumulate to a noticeable

performance degradation. This type of performance overhead clearly depends on the characteristics of the load.

An empirical study was performed to evaluate hypervisor overhead, given different types of load. Standard benchmarks were repeatedly carried out with and without an active hypervisor and performance results were recorded and compared. Tests were based on a variety of benchmarks selected from the well-known "***Phoronix Test Suite***" benchmark suite [97]. The benchmark test results are recorded in TABLE 5.

TABLE 5 Hypervisor overhead comparison results of various Phoronix benchmarks

| Benchmark | Units | No hypervisor | Thin-hypervisor | vmware | KVM |
|---|---|---|---|---|---|
| Parallel BZIP2 Compression | sec | 26.58 | 26.92 | 28.92 | 28.39 |
| Unpack Linux | sec | 10.31 | 11.81 | 14.83 | 11.4 |
| X11 - 500px Putimage Square | Op/sec | 2822 | 2795 | 1643 | 905 |
| X11 - Scrolling 500x500 px | Op/sec | 8140 | 7683 | | |
| X11 - Char in 80-char line | Op/sec | 11966667 | 10546667 | | |
| X11 - PutImage XY 500x500 Square | Op/sec | 123.73 | 120.7 | | |
| X11 - Fill 300x300px Trapezoid | Op/sec | 220500 | 210200 | | |
| X11 - 500px Copy From Win To Win | Op/sec | 6832 | 6672 | | |
| X11 - 500px Compositing: Pixmap To Win | Op/sec | 9087 | 8481 | | |

FIGURE 19 depicts the calculated overhead incurred by each type of hypervisor for each benchmark. It can be seen that the thin-hypervisor overhead is in the order of 10% or less and that it generally incurs less overhead than full-virtualization hypervisors.

### 4.5.2 Execution Protection Overhead

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, as depicted in FIGURE 15, when: (a) execution of a write-only page is attempted or (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 SLAT needs to be updated. In single-processor environments, updating the SLAT 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 SLAT. The (a) intercept, mentioned above, occurs when an executable page is first executed after the application was loaded. Both (a) and (b) intercepts occur after an executable page was swapped out and then back in. Therefore, overhead is also closely related to the swap activity in the system.

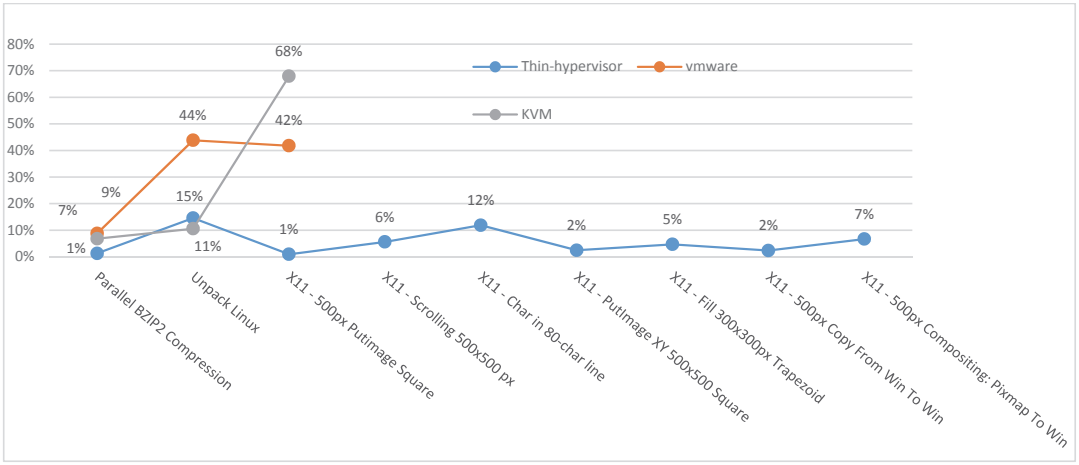


FIGURE 19 Hypervisor Performance Overhead Comparison

Empirical measurement of the performance hits that should be expected when activating the thin-hypervisor for execution protection purposes, were accomplished by two main approaches.

First, the time required for the VM exit service routine to perform a page authentication was measured directly, by fabricating a function call that generates this situation and repeatedly calling the function. Clock cycles required to complete the function call were measured and recorded. Average results show that execution-page authentication took roughly **60,000** CPU cycles. Note, that while this number is significant in itself, the number of times it occurs during normal execution of an application is extremely small. Furthermore, most occurrences happen in tandem with fetching a swapped-out page from disk, which takes an order of magnitude more than the authentication. Therefore, the associated performance overhead is generally overshadowed by other, related, performance degradations.

The second approach to measuring performance degradation was based on measuring standard benchmark results on systems with and without execution protection and comparing these results to yield average performance degradation figures under different loads. Measurements were performed on single core systems and multiple-core systems to compare the effects on different types of platforms.

A subset of benchmarks from the standard Phoronix benchmark suite [97] were executed and their results recorded (see TABLE 6) under the following conditions: (a) Single core system: (a1) free-running vs. (a2) with execution protection; (b) Multi-core system: (b1) free-running vs. (b2) with execution protection and (b3) free-running with hypervisor active. The purpose of the (b3) measurement was to isolate the performance degradation pertaining only to the hypervisor.

TABLE 6 Comparative measurements of standard benchmarks with and without execution protection

| | | | X11 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Appache Static Web Page Serving | PutImage Square | Scrolling 500 x 500 px | X11 - Char in 80-char line | PutImage XY 500 x 500 Square | Fill 300 x 300px Trapezoid | 500px Copy From Window To Window | Copy 500x500 From Pixmap To Pixmap | 500px Composit. From Pixmap To Window | 500px Composit. From Window To Window | Unpacking the Linux Kernel |
| | | Req./Sec | Op./sec | Op./sec | Op./sec | Op./sec | Op./sec | Op./sec | Op./sec | Op./sec | Op./sec | Seconds |
| Single Core | Free-run | 7978.41 | 1628 | 8140 | 10126667 | 119.43 | 147833 | 8050 | 8060 | 7180 | 6838 | 14.74 |
| | Protected | 5445.5 | 1353 | 6127 | 8141667 | 106.33 | 138250 | 7325 | 8060 | 6842 | 6537 | 15.86 |
| Multiple Cores | Free-run | 15447.43 | 3327 | 8140 | 11966667 | 123.73 | 220500 | 6832 | 8050 | 9087 | 8833 | 10.4 |
| | Protected | 10450.99 | 2465 | 7633 | 10433333 | 116.72 | 208667 | 6638 | 8047 | 8253 | 8833 | 12.73 |
| | | | | | | | | | | | | |
| | HV Free-run | 10775.58 | 2537 | 7683 | 10546667 | 120.7 | 210200 | 6672 | 8047 | 8481 | 8833 | 12.39 |

This measurement can then be subtracted from the general overhead measurement to determine the net average overhead of execution-protection. FIGURE 20 depicts the total execution protection overhead for single-core and multi-core platforms. FIGURE 21 exhibits the net execution protection overhead, obtained by deducting the hypervisor-only overhead from the general execution protection overhead. Results show that execution protection activities cause a performance degradation lower than 6%. Activating execution protection

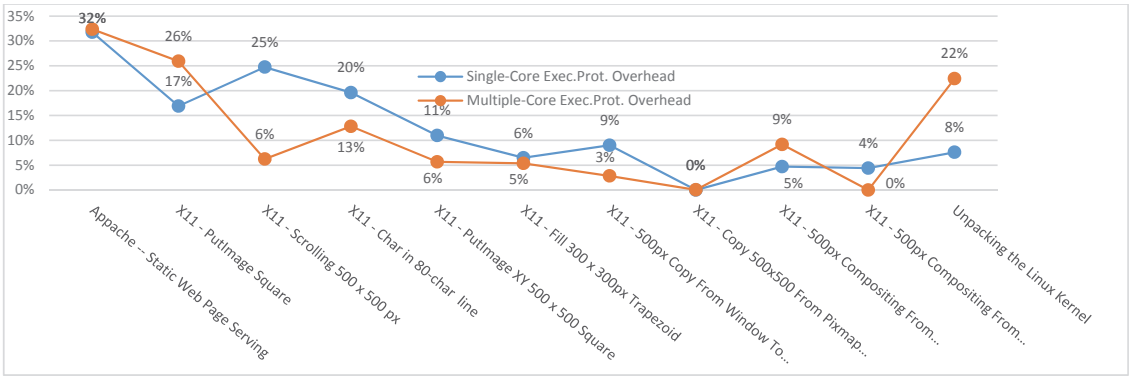


FIGURE 20 Execution Protection Overhead

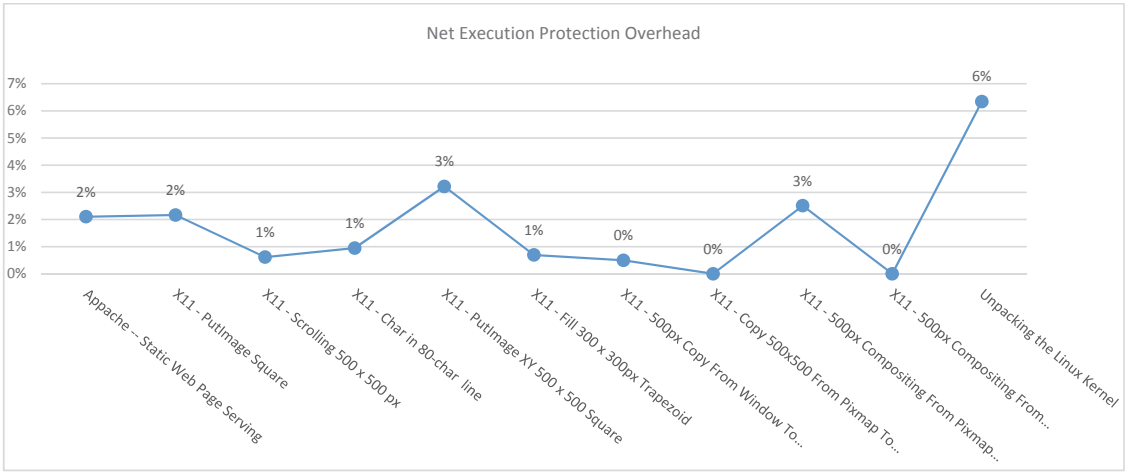


FIGURE 21 Net Mutli-Core Execution Protection. Overhead of total execution protection less overhead of an idle hypervisor.

## 4.6 Execution Protection of Interpreted Code

The execution-protection methodology described thus far is appropriate for all native-code software applications. When considering software that executes by interpretation or is based on JIT (Just in Time) compilation, this methodology needs to be augmented. The reason for this is that in the process of interpreting software, the actual machine instructions that ultimately execute are instructions that belong to the interpreter software. The interpreter itself may be non-malicious, as well as signed. Nevertheless, this does not ensure that the software it interprets is not malicious. Another way to look at this, is that interpreted software must be considered as data rather than code, since its instructions do not execute directly on the machine. Rather, they are input-data to the interpreter application.

Software applications that are based on JIT compilation similarly require special consideration since a valid signature of the software cannot be used to automatically validate the native-code instructions that will eventually be generated and then executed on the CPU.

Software applications based on interpreted code and/or JIT compilation are extremely widespread in recent years. These include, for example, Java, C#, Java-Script, Python, Ruby, Perl and PHP to name only a few frequently used languages.

The full details of the required augmentation needed in the execution-protection methodology are beyond the scope of this work. However, for completeness, an outline is presented. One possible approach to extend the concept of pre-signing executable code in the case of interpreted code is to create a chain-of-trust. The idea behind this approach is to build a methodology

layer upon layer, where each subsequent layer is deemed trusted by the previous layer. In our case the root-of-trust is the thin-hypervisor, trusted by virtue of its attestation. To support protection of interpreted or JIT-compiled software, the whitelist database shall include the signature of the (original) pre-interpreted software. Furthermore, the JIT-compiler or interpreter application shall be a pre-signed application, which includes several additions and changes relevant to the modified execution-protection scheme for interpreted code. The modifications shall delineate a procedure to follow when handling interpreted code.

The first change imposed on the Interpreter involves its preparation to execute the software. This process is the parallel of the loader's operation before executing a native-code application. The Interpreter will involve the hypervisor in preparations to run the software by making appropriate hyper-calls. In response, the hypervisor will validate the signatures of the original software. Furthermore, the hypervisor will set the access rights of all memory-pages containing the software to ***Read-Only***. The motivation for this is to ensure that any changes attempted to that memory shall trigger a VM exit intercept to the hypervisor. If this occurs, the hypervisor will set the page's access rights to ***Write-Only***, thus allowing the page to be rewritten. However, since now the page cannot be read, an attempt to interpret and execute its commands, ensures a hypervisor intercept, giving it the opportunity to revalidate the page's signature before changing its access to ***Read-Only***.

An additional change required in the Interpreter is validation of pages from which it will interpret code. Besides having ***Read-Only*** access rights, these pages must exclusively be verified pages that have undergone hypervisor validation.

The last change and the most complex one, involves isolating the Interpreter's data structures, making them inaccessible to other parts of the system – including the OS. Intel has developed the Security-Guard-Extension (SGX) technology [98], which provides Data-Enclave protection on processors that support the SGX extension. Nevertheless, such protection may also be implemented by exploiting the capabilities of the thin-hypervisor on processors that do not support SGX.

Interpreted languages that have a JIT-compiler are treated in a similar fashion. In this case, the chain-of-trust is threaded through the JIT-compiler, which is a signed application that contains several changes in support of execution-protection. The original software is validated by the hypervisor and its memory pages assigned ***Read-Only*** access rights. The JIT-compiler translates validated software pages and generates native-code in pages that the hypervisor configures ***Execute-Only*** and also creates an ad-hoc signature entry for each such page in an extension to the whitelist data-base. To ensure the chain-of-trust, the JIT-compiler's data structures must be completely isolated from the rest of the system using data-enclave technology.

# 5 MANAGEMENT STATION

## 5.1 Overview

The hypervisor prevents execution of unauthorized software by exploiting the SLAT mechanism coupled with verification of signatures in a whitelist database. Obviously, the hypervisor can do so only after its activation. Therefore, the system remains vulnerable before and during hypervisor 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 is not active and therefore cannot provide execution protection, while the system owner may be under the wrong impression that such protection is in effect.

It is, therefore, desirable to constantly inform the user in regard to the protection status of the given system. However, this notification cannot be provided by the system itself, since assuming it has been overtaken maliciously, it cannot be trusted. It is therefore self-evident that the software application that oversees the integrity a system must run on some other, remote platform, which can be guaranteed to be unaffected by the same malicious intervention. Since the Execution-Protection architecture already involves the use of a remote attestation server, which furthermore shares a secret with the attested hypervisor on the target machine, it is the perfect platform candidate to host this integrity-validation application. When envisioning a large array of computer systems, such as a network of computer stations deployed throughout the building of some enterprise, the attestation server along with the integrity-validation application can be considered an IT Management-Station for managing and monitoring the safe execution in the entire computer station network.

## 5.2 Management Station Functions

The management station, described above, has two major responsibilities: attestation and monitoring. Furthermore, it may be used as a platform to administer application software upgrades to some or all of the systems under its supervision.

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 the secret information (i.e., cryptographic key). A detailed description of this process appears in chapter 3. The attestation protocol guarantees that the secret information is provided only to authentic hypervisors, which can then protect the system (and themselves) from unauthorized use. Therefore, possession of this secret information can be regarded proof of the system's authenticity and integrity.

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 each of the remote systems, for example by displaying their statuses on the screen and sending alerts if a problem is detected. To support this scheme, the hypervisor is obligated to send a periodic message to the management station, thus indicating that the remote system is protected. This periodic message acts as a heart-beat, allowing the management system to follow-up on the well-being and activity of the hypervisor on each remote system.

To prove that its authenticity and can thus be trusted, the hypervisor signs its messages with the secret key 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. Then in each periodic message, sent ***t*** time units after the attestation has completed, the hypervisor generates and 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 22 depicts the described protocol.

## 5.3 Updating Software Applications

Once a system is whitelisted and execution-protected, i.e., it contains a data-base of signatures for all the executable pages of all its executable software applications, DLLs and drivers as well as contains and activates the thin-hypervisor – the system is limited to the exclusive execution of whitelisted software only. Execution of anything else will be immediately rejected by the hypervisor.

When new software packages need to be installed on the target computer they must be signed (on a page by page basis) and the signatures need to be added to the target-system's whitelist data-base before the new software can run. Software package installations normally comprise of a compressed archive and installation script, which guides an installation process. The installation archive may be passive, in which case it contains an installation application that the installer needs to execute; or it can be active, in which case it is a self-expanding archive the automatically expands and executes the installation script. In both cases, the first executable to run would be the installation application(s), later followed by the rest of the executable components, after the new software application launches.

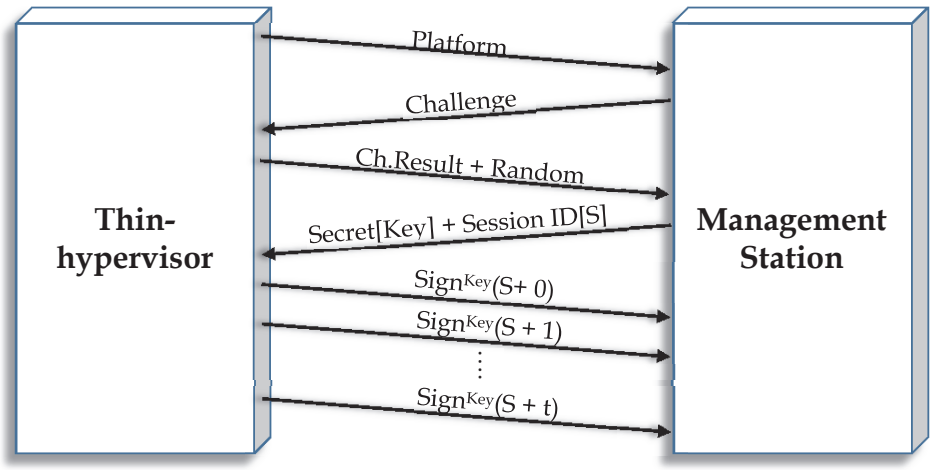


FIGURE 22 Protocol between thin-hypervisor and management station. The protocol begins with the 4-way handshake initially performed to attest the hypervisor and furnish it with secret information. It is followed by periodic notification from the hypervisor to prove that it is continuously functioning, and therefore the system can be considered protected.

Generating an extension to the whitelist data-base must be performed by the system that possess RSA signature capability. In other words it owns the RSA ***private key*** corresponding to the ***public key*** used by the thin-hypervisor for validating the signatures. Most naturally this would be the management station, since the thin-hypervisor has already made use of this public key during the attestation phase. It is proposed to install an installation archive scanner on the management station to prepare signature extensions for the target whitelist data base.

The assumption underlying a utility to generate the whitelist extension is that the installation package for the new software is verified and validated. In other words, it may be trusted as original and does not contain malware. This assumption is usually reasonable when the software installation archive has been procured directly from a well-known, and trusted (by reputation) software

vendor. In cases where additional measures must be taken, it is possible to imagine a software package being installed in a "sandbox" computer system for extensive testing, study and possibly forensics in order to validate it as "clean" software [99] [100]. Under this assumption, the utility scans the entire archive and produces signatures for all the executable pages in all the executable components found inside the archive. The installation procedure is signed as well, ditto the executable part of a self-expanding archive. The utility produces a whitelist extension, which is added to the existing whitelist database already installed on the target system, where the new software installation is intended. Once the signature extension has been added to the target's whitelist database, the installation archive can be copied to the target and activated.

## 5.4 Protecting the Management Station

Management station protection is beyond the scope of this work. However, it shall be mentioned that management station safety and trustworthiness is critical to the execution-protection scheme and the underlying assumption is that the management station can, and is, properly protected against malicious penetration. Completely assuring the cyber-security of a single system is generally a manageable IT task. We only mention several guidelines that should be followed to achieve this:

- The Management station should not be connected to the general Internet
- The management station should be connected to the local LAN behind a Firewall that prevents all possible communications, save one single TCP Listen port, across which the management station accepts TCP connections from target systems for attestation and monitoring purposes
- The management station should have a single administrator account with strong password protection
- Management station databases should be encrypted on disk and all access restricted by passwords
- Relevant passwords shall not be stored in plaintext anywhere in the management station

# 6 SUMMARY OF ORIGINAL ARTICLES

This thesis includes 7 articles. In this chapter each article is summarized and the author's contribution is stated.

## 6.1 Preventing Execution of Unauthorized Native-Code Software

Resh, A.; Kiperberg, M.; Leon, R.; Preventing Execution of Unauthorized Native-Code Software. To be published in: *JDCTA, International Journal of Digital Contents Technology and its Applications*, 2016.

This article describes the methodology underlying the prevention of malicious native-code from executing on a computer system. The paper studies the implications of implementing a thin-hypervisor, creating a trusted environment with remote attestation and managing system security by controlling memory access at a privilege level above the operating system.

The article focuses on attestation of a remote thin-hypervisor on a modern computer platform that contains multiple cores, creating a trusted environment that includes a secret key, maintaining trust and exploiting hardware virtualization to control a system's native-code execution rights with a whitelist database. The article also studies the effects of the thin-hypervisor and execution protection on system performance.

The author was a major party in the research efforts preceding this paper and was a main contributor in formalizing the algorithms and methodologies employed. The author implemented the remote-attestation and symmetric cryptography in the hypervisor side and the challenge generator in the server side, as well as took part in other implementation aspects. The author defined and supervised the performance measurements and statistical analysis related to this paper.

The author was the lead-writer of this article with research and implementation in close collaboration with the co-authors.

## 6.2 System for Executing Encrypted Native Programs

Resh, A.; Kiperberg, M.; Leon, R.; Zaidenberg, N.J.. System for Executing Encrypted Native Programs. *Reports of the Department of Mathematical Information Technology, Series B. Scientific Computing, No. B 12/2016, ISBN 978-951-39-6810-6,* 2016.

Computer software is susceptible to reverse-engineering even when it is distributed as native machine code. Reverse engineering allows an adversary to steal algorithms or administer changes to the software in an effort to bypass its licensing or administer malicious code. Solutions to combat reverse-engineering are generally based on obfuscation techniques. However, these have been proved mostly ineffective.

This article studies a new and novel methodology to create an environment that supports execution of encrypted code as a means to evade software reverse-engineering. The article focuses on the use of a thin-hypervisor to perform run-time decryption of native-code instructions, inside the CPU cache, thereby isolating the actual code from any potential adversary. The article describes the means for setting up a trusted hypervisor that contains secret key material, the hypervisor's attestation and a software encryption tool. It compares two alternative approaches to performing secure, just-in-time, decryption and execution: ***In-place*** execution vs. ***Buffered*** execution; and displays comparative performance results.

The author was a major party in the research efforts preceding this paper and was a main contributor in formalizing the algorithms and methodologies employed. The author implemented the encrypting utility and the run-time decryption and execution parts in the thin-hypervisor. The author also developed the hypervisor side AES cryptography. The author defined and supervised the performance measurements and statistical analysis related to this paper.

The author was the lead-writer of this article with research and implementation in close collaboration with the co-authors.

## 6.3 Remote Attestation of Software and Execution-Environment in Modern Machines

Kiperberg, M.; Resh, A.; Zaidenberg, N.J.. Remote Attestation of Software and Execution-Environment in Modern Machines. *The 2nd IEEE International Conference on Cyber Security and Cloud Computing*, 2015.

This article studies the problem of remotely authenticating a target system encompassing a modern hardware platform. Authentication of a remote system

is generally possible using hardware or software. The article focuses on a software only methodology and expands on existing attestation schemes to adapt to modern computer platforms that include multiple core processors and hardware virtualization. The article extends the notion of attestation to validating that the remote target system is running the correct version of software, that the system is not being virtualized and that a by-product of the attestation includes transfer of secret key material to the target system.

The author was a major party in the research efforts preceding this paper and was a main contributor in formalizing the algorithms and methodologies employed. The author led the research related to the correct exploitation of hardware side effects in modern multi-processor systems. The author implemented the remote-attestation and the challenge generator.

The author was a principle-writer of this article with research and implementation in close collaboration with the co-authors.

## 6.4 Timing and Side Channel Attacks

Zaidenberg, N.J.; Resh, A.. Timing and Side Channel Attacks. *Cyber Security: Analytics, Technology and Automation, vol. 78, pp. 183-194*, 2015.

This book chapter surveys an assortment of indirect attacks designed to break codes, guess passwords and gain illegal access into protected systems. As opposed to direct attack techniques, indirect attacks collect evidence by monitoring an abundance of parameters that are indirectly effected by the main activities. The chapter describes timing, power measurement and hardware event counters in this context.

The author was a principle-writer of this article in close collaboration with the co-author.

## 6.5 Trusted Computing and DRM

Zaidenberg, N.J.; Neittaanmäki, P.; Kiperberg, M.; Resh, A.. Trusted Computing and DRM. *Cyber Security: Analytics, Technology and Automation, vol. 78, pp. 205-212,* 2015.

This book chapter studies the concepts supporting the notion of trusted-computing and the solutions it provides to Digital Rights Management (DRM). The chapter focuses on the Trusted Platform Module (TPM) for computer system platforms. TPM methodologies are described and its weaknesses analyzed. Additional techniques and methodologies are presented as well in the areas of video and DVD rights management, as well as disk encryption and mobile phone data privacy.

The author took part in the related research and worked in close collaboration with the co-authors.

## 6.6 Can keys be hidden inside the CPU on modern Windows host

Resh, A.; Zaidenberg, N.J.. Can keys be hidden inside the CPU on modern Windows host. *ECIW 12th European Conference on Information Warfare and Security, Jyväskylä*, 2013.

This conference paper studies the alternatives and implications of storing cryptographic keys in a computer system. Alternative solutions are discussed and evaluated. The paper concludes that the major obstacle to safeguarding keys in a computer system are rogue kernel-mode drivers that may infiltrate modern operating systems. This paper's conclusions eventually led to the research and development of thin-hypervisor based techniques to solve this problem.

The author was the main party in the research that led to the findings in this paper.

The author was the lead-writer of this article in close collaboration with the co-author.

## 6.7 System for Executing Encrypted Java Programs

Kiperberg, M.; Resh, A.; Algawi, A.; Zaidenberg, N.J.. System for Executing Encrypted Java Programs. *38th IEEE Symposium on Security and Privacy (IEEE S&P 2017),* Submitted. *3rd International Conference on Information Systems Security and Privacy (ICISSP 2017),* 2017.

In recent years managed execution environments have gained increasing popularity. This article studies the applicability of protecting code against reverse-engineering by executing encrypted code in managed environments. While native machine code can be guaranteed to execute atomically in the confines of the CPU (cache for example), this guarantee cannot be extended to managed-code, since software instructions are not executed directly. Rather they need to be interpreted locally by the managing system.

This article focuses its study on the methodology required to execute managed JAVA software by incorporating parts of the JVM engine (which interprets JAVA bytecode during runtime) inside a trusted thin-hypervisor. It details the scheme for generating and attesting a trusted and secure hypervisor, which contains a secret decryption key, decrypting JAVA bytecode securely within the thin-hypervisor and the intercommunications with the parts of the

JVM outside the thin-hypervisor. Measurements of the associated performance degradation are presented as well.

The author was a major party in the research efforts preceding this paper and was a main contributor in formalizing the algorithms and methodologies employed. The author implemented the Java encrypting software.

The author was a principle-writer of this article in close collaboration with the co-authors.

# 7 CONCLUSIONS

This chapter summarizes the contributions made by the thesis, its extensions and modifications to existing practice, its limitations and direction for future research.

## 7.1 Contributions

The thesis' main contribution is a complete solution and methodology to protect a system against malicious penetration with native-code. The methodology combines several components, some previously studied and others new, novel, solutions. It also provides extensions and modifications to existing practice that were required to support modern computer platforms. To support resistance against system penetration, a remote trusted environment is setup and maintained, therefore, the contribution of the thesis is not limited to penetration-protection, but also extends to the field of trusted computing.

Hypervisors, based on hardware virtualization have become common in the last decade for the purpose of managing several operating-system stacks on a single computer platform. Thin-hypervisors, controlling only a single operating-system guest, with minimal interference, have also been proposed for a variety of purposes. The thesis expands on these ideas by extending the concept of a thin-hypervisor to execution-protection of a guest.

Creation of a remote trusted environment requires attestation capabilities of arbitrary computer platforms. The thesis expands on attestation schemes proposed by Kennell and Jamieson [8] as well as others [9] and adapts them to modern computer platforms, which contain multiple core CPUs and may themselves be operating stealthy hypervisors. The thesis also proposes how to maintain trust on a remote system following a successful attestation.

To conclude, the thesis advances the idea of resistance to malicious native-code penetration, both at the user-application level and the operating-system level, by harnessing the thin-hypervisor's control over the computer system's memory access.

## 7.2 Limitations & Future Research

Execution-protection works well for the vast majority of applications executing over Windows, Linux and OSX operating systems, running on Intel or AMD processors. However, in a small minority of executable applications, special pages exist, as described in detail in chapter 4.4. While operative solutions have been given to these singularities, a performance toll exists in most cases. Exceptional to this are self-modifying applications that modify their own code-page while executing. As mentioned in section 4.4.4, a general solution for these executable pages does not yet exist. An example to this is Microsoft's ***Patch-Guard*** executable. The specific, individual, solution given by the thesis to this situation is not completely safe. However, the barrier to exploiting it as a vulnerability is extremely high. Fortunately, these are very rare and at least in the case of Microsoft's ***Patch-Guard,*** the safest recommendation is to disable it altogether, since when execution-protection is in operation it is not needed.

The thesis methodology covers execution-protection against malicious penetration with native-code. Many modern software applications, frequently in use, are based on languages that are interpreted and/or JIT-compiled. To accommodate protection against penetration based on this type of executable software, the thesis methodology must be expanded to implement a chain-of-trust approach, as well as combine data-enclave solutions. Chapter 4.6 includes an outline for this scheme. However, it remains beyond the scope of this thesis and is left for future research.

Finally, the thesis methodology only addresses penetration attempts based on executing native-code that has been injected into the system maliciously. Granted this is the vast majority of cyber-based penetration. However, additional techniques do exist. For example, the thesis methodology cannot protect against a penetration tactic, which manages to inject data into the computer system and by virtue of that manipulate legitimate code into acting maliciously.

# YHTEENVETO (FINNISH SUMMARY)

## Laiteläheisen tietoturvan vahvistaminen

Liike-elämä on tullut entistä riippuvaisempi tietokonejärjestelmistä, niiden toiminnoista ja tietokannoista. Myös haittaohjelmien hyökkäykset ovat tulleet jokapäiväisiksi. On olemassa monenlaisia hyökkäysvektoreita, ja tietoturvallisuusala tuottaa lukuisia käyttäytymismalleihin pohjautuvia menetelmiä ilmiön havaitsemiseksi ja käsittelemiseksi.

Nykyaikaiset prosessorit jotka soveltuvat hyvin hypervisorien käyttöä tukevaan laitteistovirtualisointiin mahdollistavat useiden virtuaalikoneiden (VM) suorittamisen yhdellä tietokonealustalla. Laitteistovirtualisointitoiminnot antavat hypervisorille laitteistoalustakontrollin käyttöoikeustasolla joka ylittää käyttöjärjestelmän äyttöoikeustason.

Tämän työn tarkoituksena on tutkia ja kehittää yksinkertaistetun thin-hypervisorin pohjalta metodologia, jossa käytetään hyväksi laitteistovirtualisoinnin hyviä puolia pahantahtoisen tunkeutumisen estämiseksi tietokonesysteemiin. Onnistuminen tässä edellyttää, että luottamus yksinkertaistettuun thin-hypervisoriin täytyy olla taattu sen käskyjen, määritysrakenteiden ja sen todellisen laitteistoalustakontrollin suhteen. Lisäksi sen täytyy kyetä suojautumaan koodin turmelemiselta kaikkina aikoina. Tässä esitetty metodologia kuvaa sitä, kuinka voidaan pystyttää luotettava yksinkertaistettu thin-hypervisor ja käyttää sitä niin että se rajoittaa koodin suorittamisen yksinomaan etukäteen allekirjoitetuille, sallituille ohjelmistoille.

Tämä metodologia antaa vastustuskyvyn useimpien APT hyökkäysvektoreiden torjumiseksi ja käsittelee myös ne nollapäivähaavoittuvuudet jotka voivat jäädä huomaamatta käyttäytymismalleihin pohjautuvilta ilmaisimilta.

## REFERENCES


[1] M. Riley, B. Elgin, D. Lawrence and C. Matlack, "Missed Alarms and 40 Million Stolen Credit Card Numbers: How Target Blew It", Bloomberg, 2014.

[2] K. ZETTER, "Four Indicted in Massive JP Morgan Chase Hack", Wired, 2015.

[3] L. A. Wong, "Anthem hacked, millions of records likely stolen", CNBC, 2015.

[4] J. Pagliery, "Premera health insurance hack hits 11 million people," CNN, 2015.

[5] Wikipedia, "Ashley Madison data breach", Wikipedia, 2015.

[6] C. McCormack, "Five Stages of a Web Malware Attack," 2014. [Online]. Available: https://www.sophos.com/en-us/medialibrary/pdfs/other/5-stages-of-a-web-malware-attack-infographic.pdf.

[7] K. M. Khan and Q. Malluhi, "Establishing Trust in Cloud Computing," IT Professional, vol. 12, no. 5, pp. 20-27, 2012.

[8] 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.

[9] 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, ser. SOSP '05, New York, NY, USA, 2005.

[10] A. Seshadri, A. Perrig, L. v. Doorn and P. Khosla, "SWATT: softWare-based attestation for embedded devices," in IEEE Symposium on Security and Privacy, 2004. Proceedings., 2004.

[11] D. Schellekens, B. Wyseur and B. Preneel, "Remote Attestation on Legacy Operating Systems with Trusted Platform Modules," Sci. Comput. Program., vol. 74, no. 0167-6423, pp. 13-22, 2008.

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

[13] Brian, "Nintendo starting to ban pirates from online services on 3ds", Sony consumer electronics, 2015.

[14] M. Merritt, "Method of authenticating a terminal in a transaction execution system". Patent US5475756, 1995.

[15] S. Pearson, Trusted Computing Platforms: TCPA Technology in Context, Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[16] P. a. L. B. England, J. Manferdelli, M. Peinado and B. Willman, "A Trusted Open Platform," Computer, vol. 36, no. 0018-9162, pp. 55-62, 2003.

[17] S. Mauw, F. Massacci, F. Piessens, D. Schellekens, B. Wyseur and B. Preneel, "Special Issue on Security and TrustRemote attestation on legacy operating systems with trusted platform modules", Science of Computer Programming, vol. 74, pp. 13-22, 2008.

[18] S. Pearson, Trusted Computing Platforms: TCPA Technology in Context, Upper Saddle River, NJ: Prentice Hall PTR, 2002.

[19] J. Robertson, "Supergeek pulls off 'near impossible' crypto chip hack", NZHerald, 2010.

[20] L. Constantin, "BitLocker encryption can be defeated with trivial Windows authentication bypass", IDG News Service, 2015.

[21] C. TARNOVSKY, "Hacking the Smartcard Chip," in Blackhat, Las-Vegas,NV;USA, 2010.

[22] "Themida," [Online]. Available: http://www.oreans.com/,.

[23] "VMProtect," [Online]. Available: http://vmpsoft.com/.

[24] C. Collberg and J. Nagra, Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection, Addison-Wesley Professional, 2009.

[25] E. E. Ogheneovo and C. K. Oputeh, "Source Code Obfuscation: A Technique for Checkmating Software Reverse Engineering," International Journal of Engineering Science Invention, vol. 3, no. 51, pp. 1-10, 2014.

[26] R. Rolles, "Unpacking virtualization obfuscators," in Proceedings of the 3rd USENIX conference on Offensive technologies, Berkeley, CA, USA, 2009.

[27] L. Bohne, “Pandora’s Bochs: Automated Unpacking of Malware”, 2008.

[28] F. Gabriel, "Deobfuscation: recovering an OLLVM-protected program," December 2014. [Online]. Available: http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html.

[29] V. Roubtsov, "Cracking Java byte-code encryption," JavaWorld, 2003.

[30] A. Yadav, "Encrypted Code Reverse Engineering: Bypassing Obfuscation," Infosec Institue, 2014.

[31] A. Cavoukian and M. Dixon, Privacy and Security by Design: An Enterprise Architecture Approach, Ontario, Canada, 2013.

[32] D. K. Taft, "Secure by Design: Developing Apps Without Flaws Takes the Right Tools," eWeek, 2013.

[33] R. E. Smith, "A Contemporary Look at Saltzer and Schroeder's 1975 Design Principles," IEEE Security & Privacy, vol. 10, no. 1540-7993,

pp. 20-25, 2012.

[34] A. S. Tanenbaum, Modern Operating Systems (3rd Edition), Prentice Hall, 2009.

[35] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts 8th Edition, John Wiley & Sons, 2009.

[36] H. M. Deitel, An introduction to operating systems (2nd ed.), Addison-Wesley Longman Publishing Co., Inc., 1990.

[37] S. Crosby and D. Brown, "The Virtualization Reality," Queue, 2006.

[38] 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, Bolton Landing, NY, USA, 2003.

[39] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung and L. Smith, "Intel Virtualization Technology," Computer, 2005.

[40] M. Kiperberg, A. Resh and N. J. Zaidenberg, "Remote Attestation of Software and Execution-Environment in Modern Machines," in CSCloud, New York, NY, USA, 2015.

[41] B. Lich, "Introduction to Device Guard: virtualization-based security and code integrity policies," Microsoft, 3 2016. [Online]. Available: https://technet.microsoft.com/en-us/itpro/windows/keep-secure/introduction-to-device-guard-virtualization-based-security-and-code-integrity-policies.

[42] R. Wojtczuk, "ANALYSIS OF THE ATTACK SURFACE OF WINDOWS 10 VIRTUALIZATION-BASED SECURITY," in Blackhat USA 2016, Las-Vegas, Nevada, USA, 2016.

[43] R. J. Creasy, "The origin of the VM/370 time-sharing system," IBM J. Res. Dev., vol. 25, no. 0018-8646, pp. 483-490, 1981.

[44] I. Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3", Intel Corporation, 2007.

[45] "AMD64 architecture programmer's manual volume 2: System programming", AMD Corp., 2010.

[46] 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," SIGOPS Oper. Syst. Rev., vol. 37, pp. 164-177, 2003.

[47] A. Kivity, Y. Kamay, D. Laor, U. Lublin and A. Liguori, "kvm: the Linux virtual machine monitor," in Proceedings of the Linux Symposium, Ottawa, Ontario, Canada, 2007.

[48] 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.

[49] B. Milewski, "Virtual Machines: Thin Hypervisor," 2012. [Online]. Available: https://corensic.wordpress.com/2012/01/03/virtual-machines-thin-hypervisor/.

[50] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono and S. Chiba, "BitVisor: a thin hypervisor for enforcing i/o device security," in Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Washington, DC, USA, 2009.

[51] 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.

[52] Ben-Yehuda, Muli; Day, Michael D.; Dubitzky, Zvi; Factor, Michael; Har'El, Nadav; Gordon, Abel; Liguori, Anthony; Wasserman, Orit; Yassour, Ben-Ami, "The turtles project: design and implementation of nested virtualization", in Proceedings of the 9th USENIX conf on Operating systems design and implementation, Vancouver, BC, Canada, 2010.

[53] J. Rutkowska, "Subverting Vista Kernel For Fun And Profit", in Black Hat USA Briefings, Las-Vegas, 2006.

[54] S. T. King and P. M. Chen, "SubVirt: implementing malware with virtual machines", in IEEE Symposium on Security and Privacy (S&P'06), 2006.

[55] R. Naraine, "Blue Pill Prototype Creates 100% Undetectable Malware", eWeek, 2006.

[56] P. F. Klemperer, "Efficient Hypervisor Based Malware Detection", 2014.

[57] J. Rutkowska, "Blue Pill Detection!", 2006. [Online]. Available: http://theinvisiblethings.blogspot.co.il/2006/08/blue-pill-detection.html.

[58] G. Ou, "Detecting the Blue Pill Hypervisor rootkit is possible but not trivial", ZDNet, 2006.

[59] C. Mitchell, Trusted Computing, Institution of Engineering and Technology, 2005.

[60] P. England, "Practical Techniques for Operating System Attestation," in Trusted Computing - Challenges and Applications, Springer, 2008.

[61] A. Corporation, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," AMD, 2010.

[62] I. Corporation, "Intel Virtualization Technology for Directed I/O, Architecture Specification," 6 2016. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/docume

nts/product-specifications/vt-directed-io-spec.pdf.

[63] A. Corporation, "AMD I/O Virtualization Technology (IOMMU) Specification," 3 2011. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/488821.pdf.

[64] N. Zaidenberg and A. Resh, "Timing and Side-Channel Attacks," in Cyber Security Analytics, Vol. 3 Cybersecurity technologies, Springer, 2014.

[65] M. Renauld, F.-X. Standaert and Veyrat-Charvillon, "Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA," in Cryptographic Hardware and Embedded Systems, 2009.

[66] S. Gueron, "Intel® Advanced Encryption Standard (Intel® AES) Instructions Set - Rev 3.01," 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/.

[67] T. Müller, F. C. Freiling and A. Dewald, "TRESOR runs encryption securely outside RAM," in Proceedings of the 20th USENIX conference on Security, San Francisco, CA, 2011.

[68] K. Mowery, S. Keelveedhi and H. Shacham, "Are AES x86 cache timing attacks still feasible?," in Proceedings of the 2012 ACM Workshop on Cloud computing security workshop, Raleigh, North Carolina, USA, 2012.

[69] 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 (SOSP '05), New York, 2005.

[70] Q. Yan, J. Han, Y. Li, R. H. Deng and T. Li, "A software-based root-oftrust primitive on multicore platforms," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, New York, NY, USA, 2011.

[71] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig and L. v. Doorn, "Remote detection of virtual machine monitors with fuzzy benchmarking," SIGOPS Oper. Syst. Rev., vol. 42, no. 0163-5980, pp. 83-92, 2008.

[72] I. Corporation, "Intel® 64 and IA-32 Arcitectures Software Developer's Manual: Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C," vol. 3A, Intel, 2013.

[73] K. K. Saluja, "Linear Feedback Shift Registers Theory and Applications," pp. 4 - 14, 1987.

[74] I. Corporation, "Data Prefetch to L1 caches," in Intel® 64 and IA-32 Architectures Optimization Reference Manual, Intel, 2016, pp. 2-41.

[75] W. Hwu and Y. N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," in Proceedings of

the 13th Annual International Symposium on Computer Architecture, Tokyo, Japan, 1986.

[76] I. Corporation, "2.1.5.2. L1 DCache," in Intel® 64 and IA-32 Architectures Optimization Reference Manual, Intel, 2016, pp. 2-17 to 2-18.

[77] Y. Zhang, N. Guan and W. Yi, "Understanding the Dynamic Caches on Intel Processors: Methods and Applications," in Proceedings of the 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing, 2014.

[78] G. I. Apecechea, T. Eisenbarth and B. Sunar, "Systematic Reverse Engineering of Cache Slice Selection in Intel Processors," IACR Cryptology ePrint Archive, vol. 2015, p. 690, 2015.

[79] Y. Yarom, Q. G. a. F. Liu, R. B. Lee and G. Heiser, "Mapping the Intel Last-Level Cache," IACR Cryptology ePrint Archive, Report 2015/905, 2015.

[80] H. E. Petersen and R. Turn, "System Implications of Information Privacy," in Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, Atlantic City, 1967.

[81] K. Thompson, "Reflections on Trusting Trust," Commun. ACM, vol. 27, no. Aug 1984, pp. 761--763, 1984.

[82] M. E. Russinovich and D. A. Solomon, "x86 Interrupt Controllers," in Windows Internals, Part 1 (6th Edition), Microsoft Press, 2013, pp. 84-99.

[83] M. E. Russinovich and D. A. Solomon, "Kernel mode heaps," in Windows Internals, Part 2 (6th Edition), Microsoft Press, 2013, pp. 212-213.

[84] Wikipedia, "Second Level Address Translation," [Online]. Available: https://en.wikipedia.org/wiki/Second_Level_Address_Translation.

[85] M. Gillespie, "Best Practices for Paravirtualization Enhancements from Intel® Virtualization Technology: EPT and VT-d," 2009. [Online]. Available: https://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d.

[86] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, San Jose, California, USA, 2006.

[87] I. Corporation, "Intel 64 and IA-32 Architecture Software Developers Manual," vol. 3A, pp. 4-1 to 4-21.

[88] R. Wilkins and T. Nixon, "The Chain of Trust: Keeping Computing Systems More Secure," 2016. [Online].

[89] L. Abrams, "How to create an Application Whitelist Policy in

Windows," BLEEPINGCOMPUTER, 2016.

[90] "Application Whitelisting for Today's Dynamic Endpoint Environment," Lumension, [Online]. Available: https://www.lumension.com/application-control-software/application-whitelisting.aspx.

[91] R. A. Grimes, "InfoWorld review: Whitelisting security offers salvation," InfoWorld, 2009.

[92] M. Pietrek, "An in-depth look into the Win32 portable executable file format - Part 2," MSDN Mag., pp. 80-90, 2002.

[93] E. Youngdale, "Kernel Korner: The ELF Object File Format by Dissection," Linux Journal, p. 15, 1995.

[94] Microsoft, "Kernel patch protection: frequently asked questions," Microsoft, 22 Jan 2007. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/Dn613955(v=vs.85).aspx.

[95] Skywing, "PatchGuard Reloaded: A Brief Analysis of PatchGuard Version 3," Uninformed, vol. 8, 2007.

[96] J. Heo and R. Taheri, "Virtualizing Latency-Sensitive Applications: Where Does the Overhead Come From?," vmware technical journal, vol. 2, pp. 21-30, 2013.

[97] P. Media, "Phoronix test suite," Phoronix Media, 2016. [Online]. Available: http://www.phoronix-test-suite.com/.

[98] I. Corporation, "Intel Software Guard Extensions (Intel SGX)," [Online]. Available: https://software.intel.com/en-us/sgx.

[99] V. Systems, "Veeva Vault Validation," 2015. [Online]. Available: https://www.veeva.com/wp-content/uploads/2015/09/Vault-Validation-Datasheet.pdf.

[100] L. Grunske, R. Reussner and F. Plasil, "Component-Based Software Engineering," in 13th Internation Symposium, CBSE 2010, Prague, Czech Republic, 2010.

[101] R. Sailer, X. Zhang, T. Jaeger and L. van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," in Proceedings of the 13th Conference on USENIX Security Symposium, Berkeley, CA, USA, 2004.

[102] I. Corporation, "Intel® Software Guard Extensions (Intel® SGX)," [Online].Available: https://software.intel.com/en-us/sgx.

# ORIGINAL PAPERS

# I

## PREVENTING EXECUTION OF UNAUTHORIZED NATIVE-CODE SOFTWARE

by

Resh, A.; Kiperberg, M.; Leon, R., Zaidenberg N.J. 2016

To be published in: *JDCTA, International Journal of Digital Contents Technology and its Applications*

# 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, The 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.

## 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 [5]. After 2005 Intel and AMD introduced hardware support for virtualization (Intel VT-X [6], AMD AMD-V [7]) 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.

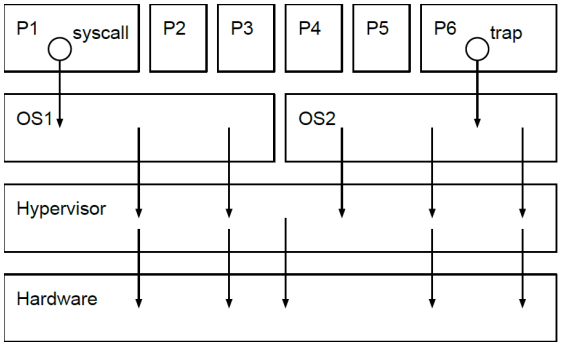


**Figure 1.** Virtualized system featuring a hypervisor and two operating systems executing 6 programs. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions and other interrupts, transfer control from user mode applications to their operating system. 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 [8] [9]. 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.

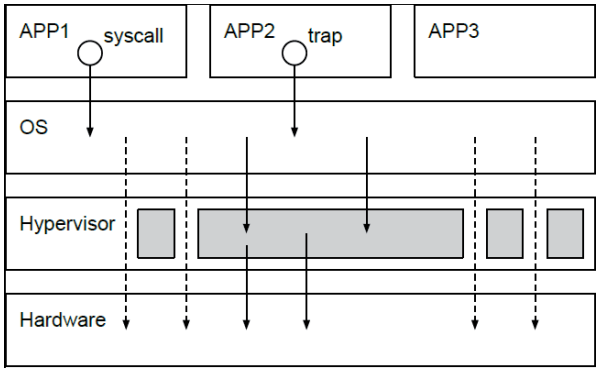


**Figure 2.** Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service 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 [10] 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 [11] 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 [12] [13] [14] [15] [16] [17]. 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 [18] [19] [20] and only authenticated bank terminals can be allowed to fetch records from the bank database [21].

The research in this area can be divided into two major branches: hardware assisted authentication [22] [23] [24] and software-only authentication [12] [13] [25]. 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 [12] 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 [11]. 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 [26] [27]. 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.

### 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.

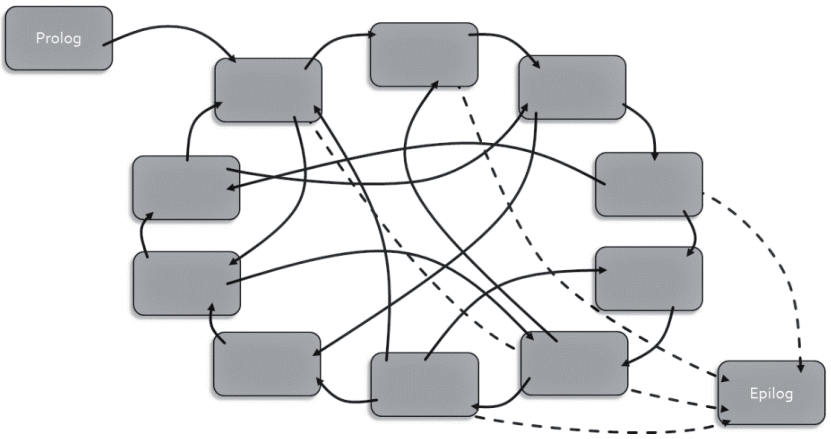


**Figure 3.** A challenge node network.

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 [28]. 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.

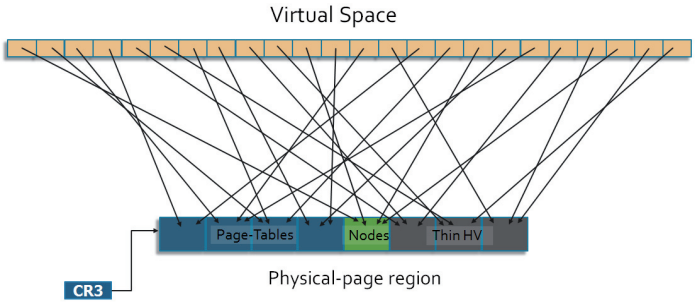


**Figure 4.** A challenge node network.

## 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) [6] and Rapid Virtualization Indexing (RVI) [7], 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 [6] 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 [29] [30]. 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 [31] or ELF file in Linux [32]) 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.

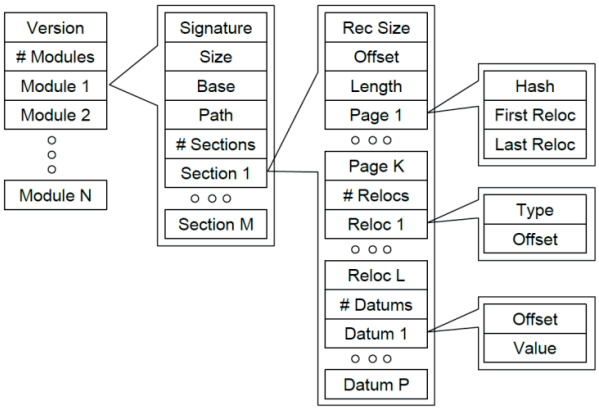


**Figure 5.** Structure of the database containing the hash values. The database consists of many modules, each of which consists of many sections. Each section contains the hash values of pages that it occupies, the relocations in those pages and datums – values of relocation that cross page boundaries.

### 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.

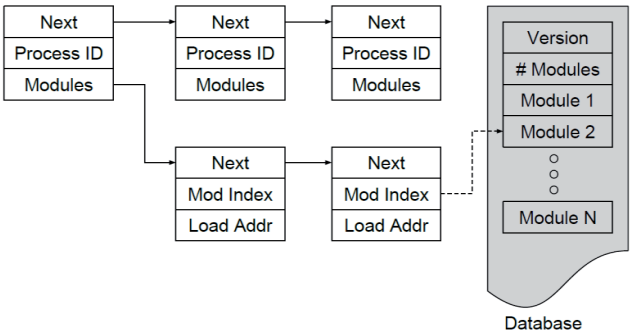


**Figure 6.** Memory layout data structure. The memory layout consists of a list of process descriptors. Each process descriptor contains the process identifier of the corresponding process and a pointer to a list of module descriptors. Each element of the module descriptors list contains the index of the corresponding module and its location in memory.

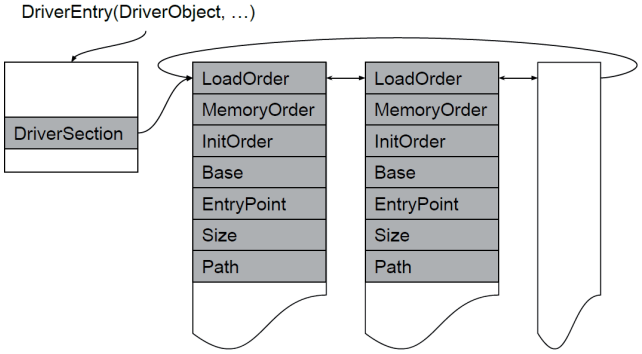


**Figure 7.** Memory layout data structure. The memory layout consists of a list of process descriptors. Each process descriptor contains the process identifier of the corresponding process and a pointer to a list of module descriptors. Each element of the module descriptors list contains the index of the corresponding module and its location in memory.

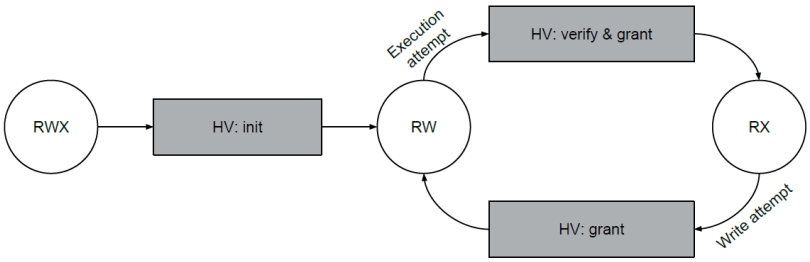


**Figure 8.** Physical pages access rights state diagram. "RWX" represents full access rights. "RW" represents "read" and "write" access rights. "RX" represents "read" and "execute" access rights.

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 [6]. 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.

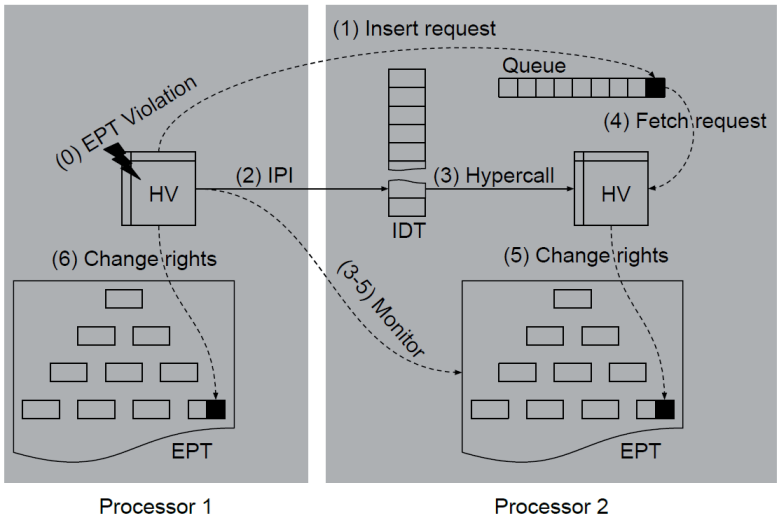


**Figure 9.** Access rights modification on a multiprocessor system: (0) an EPT violation on processor 1 triggers the hypervisor; (1) the hypervisor inserts a request into the request queue of processor 2; (2) the hypervisor sends an IPI to processor 2; (3-5) the hypervisor monitors the EPT hierarchy 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.

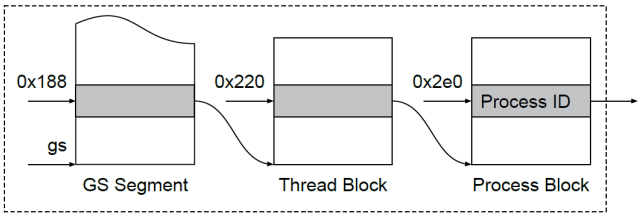


**Figure 10.** The GS register points to a local storage of the current processor. This local storage points to a data structures that represents the currently executing thread – the thread block. The thread block points to a data structure that represents the process which hosts the thread – the process 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.

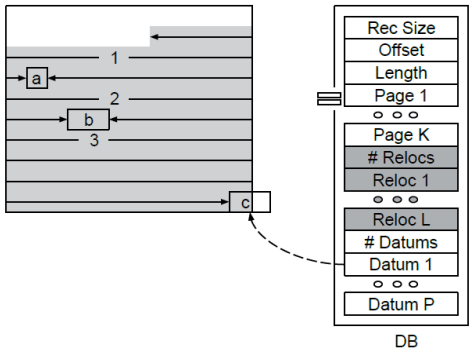


**Figure 11.** Page authentication in its most general form. In this case the section starts in the middle of a page. The section contains three relocations: *a*, *b* and *c*. Relocation *c* only partially belongs to the page being authenticated. The verification function first computes the hash of the 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.

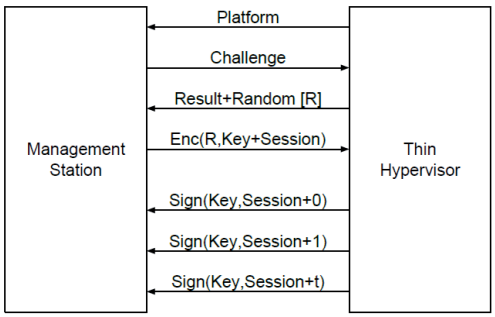


**Figure 12.** The protocol between the management station and the thin-hypervisor. The protocol consists of a 4-way handshake and periodic notifications. The "+" sign here means concatenation.

## 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" [33]. 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.

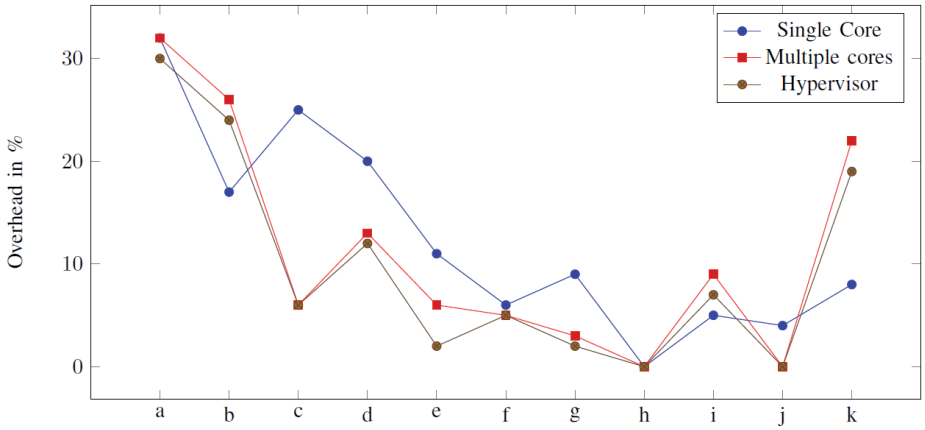


**Figure 13.** Overhead of the benchmark execution under different conditions: (a) single core; (b) multiple cores; and (c) with hypervisor but without execution protection

### 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.

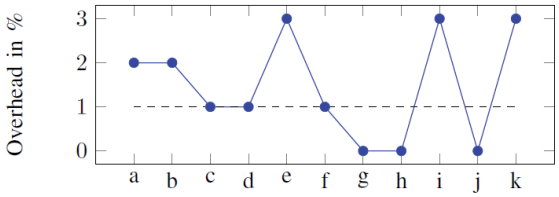


**Figure 14.** Overhead of execution protection only after subtraction of the hypervisor overhead. The dashed line represents the average overhead.

| | 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] 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.

[6] C. Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," vol. 3, 2007.

[7] AMD, "AMD64 Architecture Programmer's Manual: System Programming," vol. 2.

[8] 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.

[9] 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.

[10] 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.

[11] M. Kiperberg, A. Resh and N. J. Zaidenberg, "Remote Attestation of Software and Execution-Environment in Modern Machines," in *CSCloud*, 2015.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] Sony, "Information on banned accounts and consoles," 2015.

[20] Brian, "Nintendo starting to ban pirates from online services on 3ds," Nintendo everything, 2015.

[21] Wikipedia, "An analysis of proposed attacks against genuinity tests," [Online]. Available: http://en.wikipedia.org/wiki/Warden .

[22] 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.

[23] S. Pearson, Trusted Computing Platforms: TCPA Technology in Context, Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[24] 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.

[25] 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.

[26] 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.

[27] E. G. a. C. J. Mitchell, "Trusted computing: Security and applications," *Cryptologia,* vol. 33, no. no. 3, p. 217–245, 2009.

[28] K. K. Saluja, Linear feedback shift registers theory and applications, 1987.

[29] 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.

[30] A. Dang, "Behind Pwn2Own: Exclusive Interview With Charlie Miller," March 2009. [Online]. Available: http://www.tomshardware.com/reviews/ pwn2own-mac-hack,2254-4.html.

[31] M. Pietrek, "An in-depth look into the Win32 portable executable file format," *MSDN Mag. 17, 2,* pp. 80-90, 2002.

[32] E. Youngdale, "Kernel korner: The elf object file format by dissection," *Linux Journal,* vol. 1995, no. no. 3es, p. 15, 1995.

[33] M. Larabel and M. Tippett, "Phoronix test suite," Phoronix Media, [Online]. Available: http://www.phoronix-test-suite.com/. [Accessed June 2016].

# II

## SYSTEM FOR EXECUTING ENCRYPTED NATIVE PROGRAMS

by

Resh, A.; Kiperberg, M.; Leon, R.; Zaidenberg, N.J. 2016

To be published in: *JDCTA, International Journal of Digital Contents Technology and its Applications*

# 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, The 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.

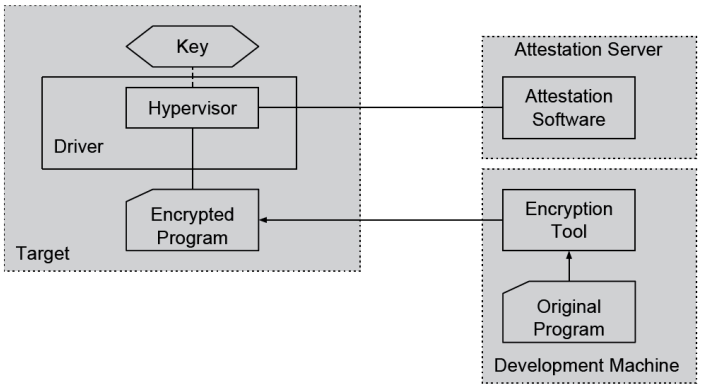


**Figure 1.** Native code protection system. The original program is encrypted before its distribution. The encryption key is stored in the attestation server, which delivers it to the hypervisor in the target machine upon successful attestation. The hypervisor is initialized by a driver, which also 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.

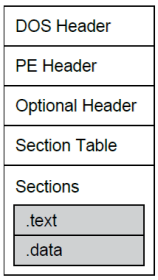


**Figure 2.** Structure of a Windows PE file. The structure contains a variable number of sections. Two of the most common sections are presented.

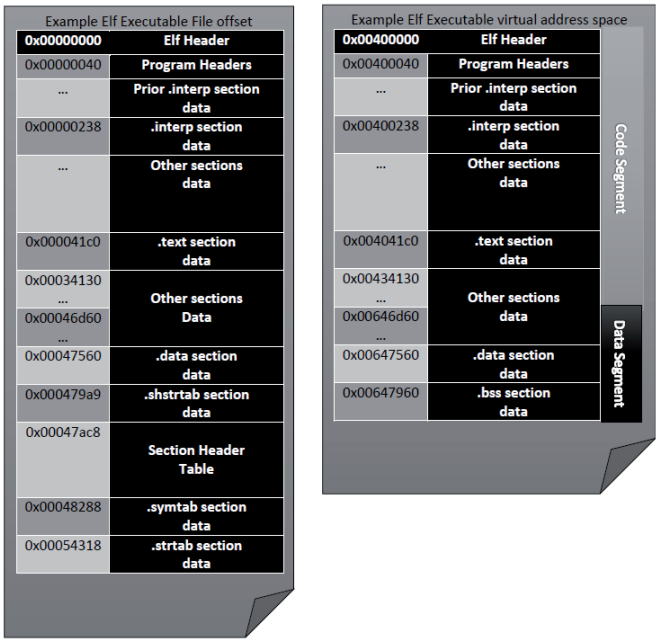


**Figure 3.** The left image represents the structure of an ELF file as it is stored in disk. The right image represents the structure of an ELF file as it is loaded to memory.

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.

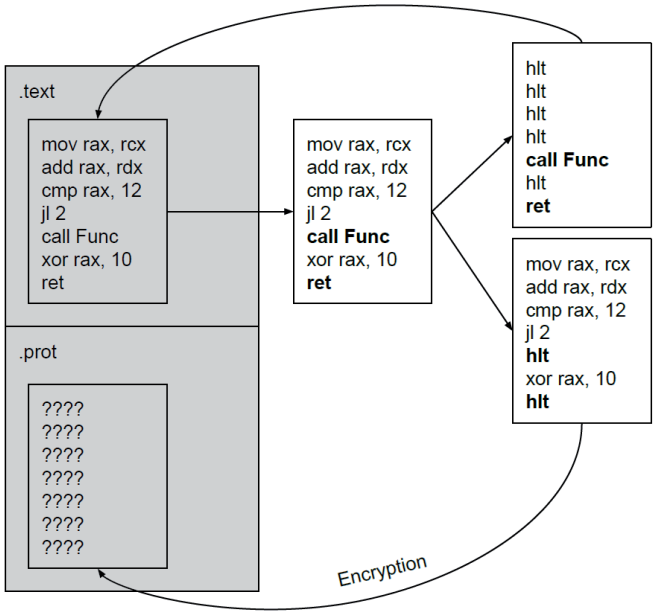


**Figure 4.** Example of an encryption process of a single function. The encryption begins by classifying instruction is encryptable (normal face) and non-encryptable (bold face), and creating to copies. The complementary instructions in each copy are replaced by halts. Finally, one copy is written over 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.

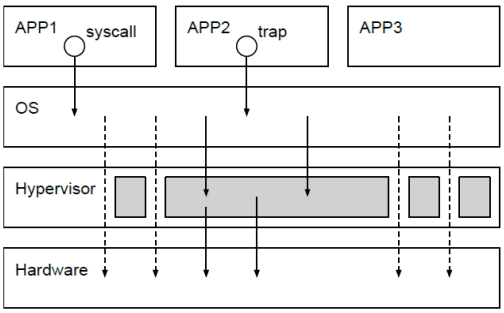

**Figure 5.** Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions, and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service 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]. 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].

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.

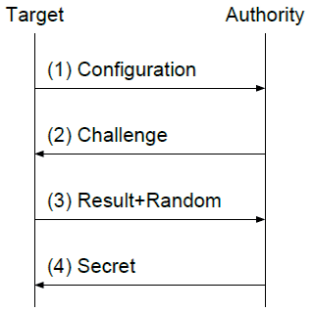


**Figure 6.** The attestation protocol between the authentication authority and the target machine. The protocol consists of four messages. The first two messages are sent unencrypted, while the two last messages are encrypted. The third message is encrypted by the public key of the authentication authority 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.

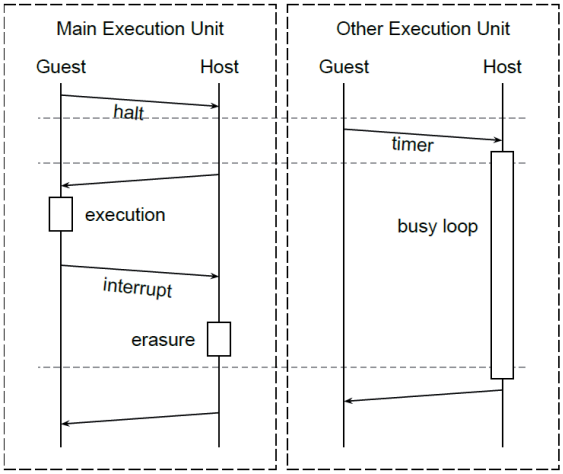


**Figure 7.** Example of encrypted function execution. The figure depicts two execution units, each with two alternating states: guest and host. The dashed horizontal lines are synchronization barriers, i.e. everything above the line is guaranteed to complete before anything below the line starts.

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.

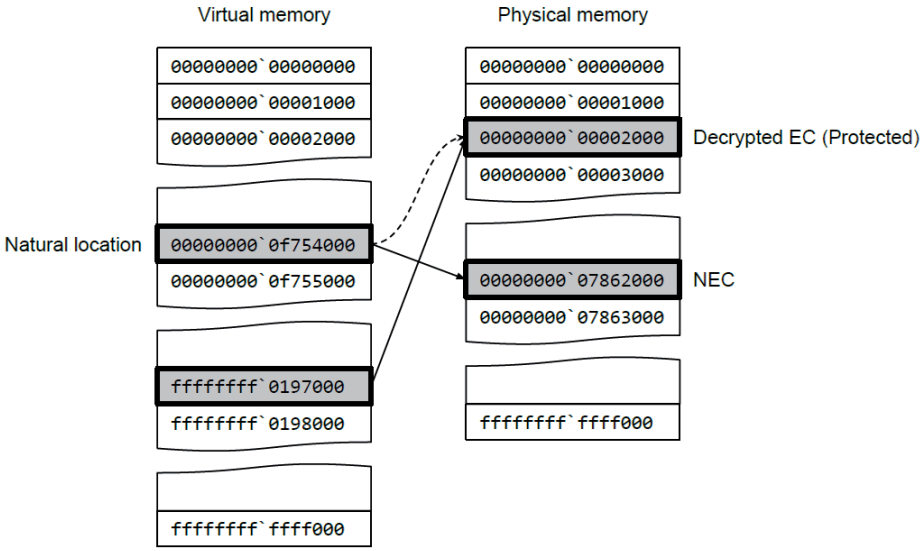


**Figure 8.** Memory layout during buffered execution. The functions resided at virtual address f754000, which is mapped to the physical address 7862000. The encrypted code is decrypted to virtual address ffffffff`0197000 which is mapped to the physical address 2000. The hypervisor changes the mapping 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.

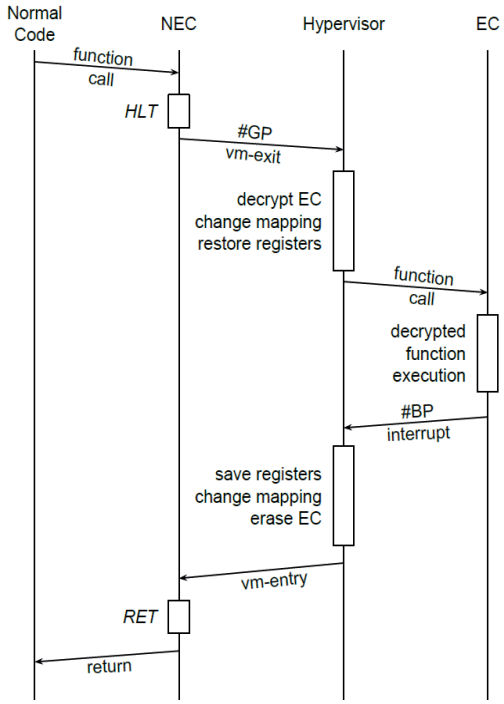


**Figure 9.** Example of encrypted function execution in buffered execution mode. The figure depicts the control flow during the execution of an encrypted function.

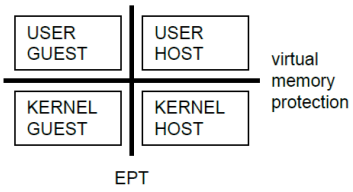


**Figure 10.** Execution modes. The left column represents the guest mode, while the right column represents the host mode. The lower row represents the kernel mode, while the upper row represents the user mode. The host mode can protect itself from the guest mode through the EPT mechanism. The kernel 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.

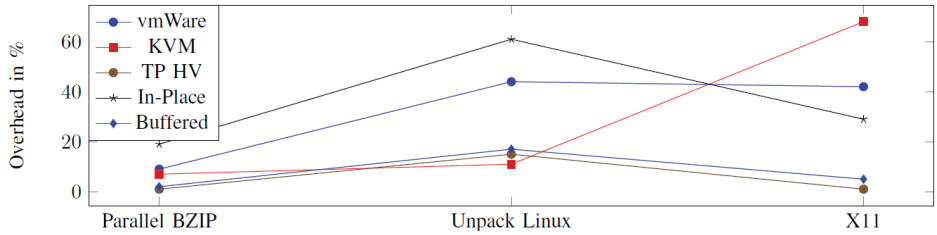


**Figure 11.** Overhead calculation relative to no-hypervisor benchmarks.

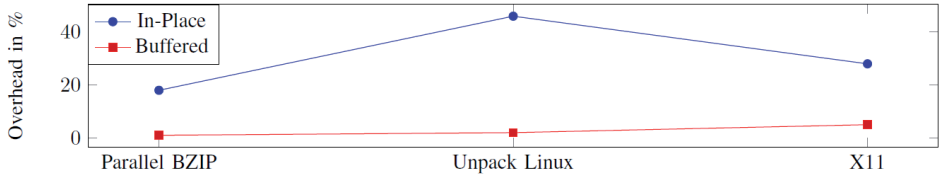


**Figure 12.** Net encrypted execution overhead.

## 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/aid/ 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

# III

## REMOTE ATTESTATION OF SOFTWARE AND EXECUTION-ENVIRONMENT IN MODERN MACHINES

by

Kiperberg, M.; Resh, A.; Zaidenberg, N.J 2015

The 2nd IEEE International Conference on Cyber Security and Cloud Computing

# Remote Attestation of Software and Execution-Environment in Modern Machines

Michael Kiperberg
Deparment of Mathematical IT
University of Jyväskylä
Finland
Email: mikiperb@student.jyu.fi

Amit Resh
Deparment of Mathematical IT
University of Jyväskylä
Finland
Email: amit.resh@jyu.fi

Nezer J. Zaidenberg
Deparment of Mathematical IT
University of Jyväskylä
Finland
Email: nezer.j.zaidenberg@jyu.fi

***Abstract*—The research on network security concentrates mainly on securing the communication channels between two endpoints, which is insufficient if the authenticity of one of the endpoints cannot be determined with certainty. Previously [1], [2] presented methods that allow one endpoint, the authentication authority, to authenticate another remote machine. These methods are inadequate for modern machines that have multiple processors, introduce virtualization extensions, have a greater variety of side effects, and suffer from nondeterminism. This paper addresses the advances of modern machines with respect to the method presented in [1]. The authors describe how a remote attestation procedure, involving a challenge, needs to be structured in order to provide correct attestation of a remote modern target system.**

## I. Introduction

Software, hardware and hybrid solutions for computer-systems security are facing modern cyber-breaches, viruses, worms, rootkits and other malicious executables. These security solutions may be implemented successfully by remotely creating a trusted application container. The application container is executed on a, possibly, already compromised system. One possible embodiment to manage this security paradigm involves an authentication authority, which can administer an authentication procedure to a remote machine.

The problem of remote software authentication, determining whether a remote computer system is running the correct version of a software, is well known [1]–[8]. 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 [9]–[11] and only authenticated bank terminals can be allowed to fetch records from the bank database [12].

The research in this area can be divided into two major branches: hardware assisted authentication and software-only authentication. While in theory, hardware assisted authentication may provide more conclusive results regarding the authenticity of a remote machine, in practice the hardware fails to provide additional security due to inappropriate designs of currently available operating systems [5].

Hardware assisted authentication uses an external hardware component, such as Trusted Platform Module (TPM) to compute a cryptographic hash of the computers hardware and software configuration and attest it.

Usually [13]–[15] the TPM is used as the root of the chain of trust. The TPM measures the authenticity of the BIOS. The BIOS then measures the authenticity of the boot loader and so on. Unfortunately, all common modern operating systems (e.g. Linux, Windows, OS X) allow the user to load drivers for execution with the same privileges as the operating system itself, i.e. ring 0 on x86 and x64 hardware. Malicious or buggy drivers, which are executed with high privileges, allow random code execution that makes it possible to circumvent the authenticity measurements of the TPM.

Software-only authentication usually targets a specific instruction set architecture that varies from ATMega [3], through Pentium [1] to Intel Core [16]. 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.

A hybrid approach, in which the TPM is used as an external source of time, was described in [5]. According to this approach the TPM is used to time-stamp the beginning and the completion of the challenge execution, thus reducing the unpredictable deviations in time caused by network delays. The TPM needs to be built on tamper proof hardware, an assumption that is not always true as was shown by Tarnovsky [17], [18].

Pioneer [2] is a software-only component designed to provide execution of a remotely authenticated executable on an untrusted and possibly compromised legacy host system. Pioneer is composed of a dispatcher system that is used to manage a challenge-response protocol with the untrusted

platform, where an authenticated executable is to be run. The methodology of Pioneer is based on a verification utility, which first establishes itself as a root of trust, by executing code that both checksums itself and verifies that it is running. The verification utility is randomized by receiving a challenge seed from the dispatcher. Once trusted, the verification utility proceeds to authenticate the executable in question.

Pioneer is based on two assumptions on the untrusted platform: it has a single logical processor, and it does not contain a virtualization extension. Logical processors multiplicity, which was introduced in modern CPUs, violates the assumptions of Pioneer. The authors propose a remedy for this vulnerability by introducing a data dependency between the different parts of the challenge [16], thus preventing its parallel execution. Pioneer execution on processors with a virtualization extension is discussed in [19]. The authors describe a modification to the original method which allows not only to achieve consistent results on all processors but also to employ intermediate variations to detect virtualized environments.

The method proposed in [1] 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, the authors of [1] conclude that it will not be able to compute the correct result within the predefined time-frame.

The conclusion is probably true for the instruction set architecture that was considered in [1]. Modern instruction set architectures allow to construct an "emulator" that performs a single native instruction for every simulated instruction. This construction is provided through a virtualization extension of modern Intel and AMD processors. In short, the virtualization extension allows the software to specify a set of events to be intercepted and a function to be called on their interception. The events can be privileged instruction execution, memory access, interrupts, etc. Since the software can alter this set at any point, it can disable interception of the events that can occur during the execution of the challenge and re-enable their interception when the challenge completes.

Another feature of modern processors that was not discussed in [1], is multi-processing. During the execution of the challenge by one logical processor, another logical processor may affect the caches, which will lead to an incorrect result. Moreover, the additional computational power of other logical processors can potentially be utilized to deceive the authentication authority.

In this paper we discuss the deficiencies of the method described in [1] on modern machines that arise from the virtualisation extension and multiplicity of logical processors. We present solutions to the discussed problems.

1) R $\rightarrow$ A: R's configuration
2) A $\rightarrow$ R: Authentication challenge
3) R $\rightarrow$ A: Result and random value
4) A $\rightarrow$ R: Sensitive information

Fig. 1. This figure depicts a possible communication protocol between the authentication authority (A) and the remote machine (R).

This paper is organized as follows. Section II presents methods of software-only remote authentication described in [1], [2]. In section III we discuss the implications of the virtualization extensions on these authentication methods. Complications that arise from logical processors multiplicity are discussed in section IV. Section V shows how side-effects information on modern processors can be obtained and fully utilized. Non-deterministic behavior of modern processors is described in section VI. In section VII we discuss the conditions that can allow an adversary to deceive the authentication method described in this paper. Section VIII concludes our results.

## II. Remote Authentication of Legacy Machines

Remote authentication methods define a protocol between the authentication (attestation) authority and the remote machine. The protocol enables the authentication authority to determine whether the remote machine is authentic.

Figure II depicts a possible structure of the authentication protocol.

The initial messages of the protocol carry information about the current configuration of the remote machine (transmitted by the remote machine). Following this exchange, the authentication authority transmits a message containing the challenge code to be executed on the remote machine. The remote machine executes the challenge, which computes a result, that is a cryptographic hash of some memory region, possibly with some additional information, and transmits it back to the authentication authority. The authentication authority verifies that the result is correct and was received within a predefined time-frame. If both are true the remote machine is considered authentic.

For practical reasons, the remote machine can generate a random number, concatenate it to the result, and encrypt both values before sending the reply to the authentication authority to avoid replay attacks. The remote machine can then use this random value, called the session key, as a proof of its authenticity.

For example, this value can be used as an encryption key to securely transmit some sensitive information from the authentication authority back to the authenticated remote machine. Clearly, an unauthenticated machine will not be sent this sensitive information.

The structure of the challenges and the hardness assumptions vary between authentication methods. Some methods [2], [3] choose the code of the challenge carefully and guarantee that the challenge constitutes the most efficient computation of the desired result. Other methods [1], incorporate side effects

into the computed result, thus, in some sense, utilizing the entire processor circuitry in result computation. The goal of both types of methods is to make it impossible to emulate execution of the challenge on a non-authentic machine within the predefined time-frame.

Another similarity between the structures of the challenges produced by both types of methods is the division of the challenge into blocks and the unpredictable control flow between the blocks. The control flow depends on the intermediate values of the result. An invalid intermediate value produces a different control flow, which in turn naturally leads to an invalid final result.

Following each authentication request, a pseudo-random challenge is transmitted, to eliminate replay attacks. The authentication authority generates the challenges and computes their results ahead of time. The blocks, their relative order and the control flow are chosen pseudo-randomly during the generation phase.

The blocks are constructed for a specific architecture. Advances in instruction set architectures can potentially render the current blocks obsolete, by allowing new types of attacks that are not prevented by the current variety of block types. The most significant advances that require special consideration are multi-core architectures and virtualization extensions.

The methods described in [2], [3] are subject to attacks on multi-core processors [16]. The additional computational resources can be utilized to deceive the authentication authority. The authors of [16] propose to mitigate this attack by widening the variety of blocks. The effect of virtualization extensions on the methods described in [2], [3] were studied in [19]. Some of the operations performed by the challenge blocks produce different results in presence of an active virtual machine monitor, thus producing an invalid final result. The authors explain not only how to accommodate this diversion, but also how this diversion can be incorporated into the computed result, thus providing the authentication authority with information regarding the configuration of the remote machine.

The structure of blocks is discussed in [1]. The blocks can be one of two types: blocks incorporating memory content and blocks incorporating side-effects. Blocks of the first type read content of memory from some pseudo-random location and incorporate it into the accumulated result. Blocks of the latter type fetch some information regarding side-effects from the processor or the environment and incorporate it into the computed result using a non-commutative calculation (with regard to blocks of the first type). For example, if blocks of the first type use addition, blocks of the second type can use exclusive-or or rotate.

Every instruction that is executed by a processor modifies its internal state. Some modifications result from the definition of the instruction operations; others — are performed by the processor to improve performance, e.g. cache population, or for debugging and profiling purposes, e.g. L3 cache miss count. Previously, processors were allowed to observe the state of side-effects directly. Current versions of processors

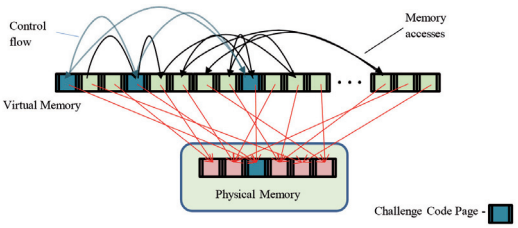


Fig. 2. The figure depicts the mapping between the virtual and the physical memories. The challenge is stored only in one page of the physical memory (the blue square) but can be accessed via many virtual pages. During the execution of the challenge the control flow is transferred between blocks in different virtual pages to utilize the ITLB. The memory is accessed by the challenge in pseudo-random order. The same data can be read multiple times via different virtual pages.

provide a different mechanism: performance counters. The processor defines pairs of registers: an event selection register, which allows the software to specify the execution event to be counted, and a monitoring counter register, which is increased on each occurrence of the event specified by the first register. The values of the counter registers can be considered the state of the side-effect and as such can be incorporated into the result.

It is desirable to construct the challenge in a way that maximizes the side-effects produced by its execution. One of the side-effects that were considered in [1] is the TLB management system. TLBs store translations of virtual addresses to physical addresses of pages that were recently accessed. Modern processors contain separate TLBs for instructions and data as well as a shared TLB of a higher level, which is larger but slower. When a new translation needs to be stored in a TLB with no free slots, one of the slots is evicted according to some policy, which varies between processors. In order to achieve high utilization of the TLBs the authors of [1] propose to map a large virtual memory region that maps a smaller physical memory region that is to be authenticated (Fig. II). The challenge then can compute the hash by reading the contents of the physical memory region through different pages of the virtual memory region, thus fully utilizing the TLBs and inducing more side-effects. A typical layout of the physical memory region that is mapped by the virtual memory is depicted in Fig. II.

## III. Virtualization Extension

Virtualization extension instructions are an extension to the x86 instruction set architecture that allows isolation of multiple operating systems efficiently, thus providing means to construct virtual machine monitors [20], [21]. Previously, construction of virtual machine monitors involved binary instrumentation and required modification in the code of the hosted operating systems.

There are slight differences between Intel's and AMD's implementation of the x86 virtualization extension. In this

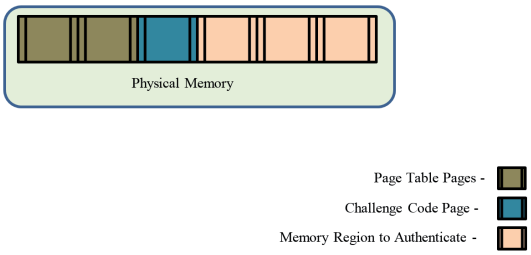


Fig. 3. The figure depicts the layout of the physical memory that is mapped by the virtual memory. The brown squares correspond the pages occupied by the virtual page tables. The challenge is stored in single blue square, which represents a single page. Other squares represent the physical memory region that is authenticated by the challenge. Note that the challenge simultaneously authenticates the contents of the entire range of physical pages: page-table, challenge-code and memory region to authenticate.

paper we will discuss only Intel's implementation and mention the differences where they are important for the discussion.

A virtual machine control structure (VMCS) is defined for each virtual environment managed by a virtual machine monitor (VMM). 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.

Interception of some events cannot be disabled, while interception of others cannot be enabled. For example, execution of the CPUID instruction always causes a vm-exit, while execution of the SYSCALL instruction never causes a vm-exit. Processors produced by AMD allow disabling interception of all events.

The existence of the VMM, does not affect the internal state of the processor. Therefore, the authentication method described in [1] can succeed in presence of a VMM. However the VMM can intercept execution of privileged instructions and modify their behavior, thus acting as malicious code. The code itself can be hidden using the physical mapping as described above.

We suggest the following method for VMM detection. Since Intel processors do not allow to disable interception of the CPUID instruction, execution of this instruction forces a vm-exit. On vm-exit, the processor loads the first instruction of the function whose address is specified in VMCS. This behavior alone will affect some of the caches, regardless of the actual implementation of the function. The lookup of the address modifies at least one entry of the ITLB and the higher level TLB (STLB). Fetching the first instruction modifies at least one entry in the instruction cache, L2 cache and L3 cache. In addition, execution of such an instruction takes much more time when a VMM is active. Therefore, we propose to widen the variety of blocks by adding blocks that produce events whose interception cannot be disabled. An example of such a block is a block that contains a CPUID instruction.

Unfortunately, on processors produced by AMD there are no such events that are guaranteed to be intercepted. However, the virtualization extension is enabled on AMD by setting a bit of a model specific register called EFER. We propose to add a block that reads the value of this model specific register and incorporates it into the result. If the bit is set in the value of this register, as perceived by the challenge, then the authentication will fail. If the bit is not set, then either a VMM is not active or it is active but it intercepts accesses to the EFER register and alters its behavior. In the latter case, however, the interception will modify the internal structure of the processor and will be detected as a result of side-effects. Thus, we force an adversary VMM to intercept accesses to the EFER register by incorporating its value in the result.

## IV. Accommodating Multiple Processors

Modern processors consist of several execution units, called logical processors, each of which contains separate execution units (instruction decoder, branch predictor, arithmetic logic unit (ALU), floating point unit, etc.). The processor has multiple units of cache memory: translation lookaside buffer (TLB) caches, instruction cache and data caches. TLB caches are used to store information regarding virtual-to-physical address translations. The caches are separated into instruction TLB and data TLB and a unified, larger TLB. The instruction and data TLBs are separate for each logical processor while the unified TLB is shared by a group of logical processors, called a core. Similarly, some caches, like the instruction cache and the L1 cache, are separate for each logical processor while the L3 is shared between all the logical processors. In addition, for simplicity, the caches obey the inclusion policy, by which the higher level caches include all the information contained in the lower level caches. This policy implies that if a line is evicted from a higher level cache it has to be evicted from all the caches beneath. Therefore any logical processor can cause eviction of data from a cache that is owned solely by another logical processor.

We can conclude from the above discussion that in order to preserve determinism of the cache memories state it is required to "freeze" all logical processors but the one executing the challenge. Unfortunately, simple solutions, like idle-loops are not sufficient, since they affect the instruction TLBs and the instruction caches.

```
01. static step = 0;

02. WaitFor(value):
03.   forever:
04.     MONITOR step
05.     if step=cur_proc break
06.    MWAIT step

07. ScheduledFunc(cur_proc,
              total_procs, challenges):
08.   step <- step + 1
09.   wait_for(total_procs + cur_proc)
10.   execute challenges[cur_proc]
11.   encrypt the result
12.   step <- step + 1
13.   WaitFor(2 * total_procs)
```

Fig. 4. Pseudo-code of the challenge execution for multiple logical processors

We suggest to use the MONITOR/MWAIT pair of instructions to "freeze" other logical processors during challenge execution. These instructions take a specified memory range and put the processor in an idle state until the contents of that specified memory region is modified. Since no instructions are executed, the caches are not affected by the idle logical processors.

The pseudocode of our solution is given in Fig. IV. The protocol is executed by one of the logical processors, which receives the challenge and schedules the routine *Scheduled-Func()* (line 07) on all logical processors. The logical processors use the static variable *step* (line 01) for synchronization. Only two manipulations are performed on this static variable: increment and comparison. The comparison is performed by the routine *WaitFor()*, which loops until the value of the static variable *step* becomes *value*.

The loop uses the MONITOR and MWAIT instructions that block until the specified memory region is written by some other logical processor. Consider the execution of *Scheduled-Func()* by the $N$th logical processor. Line 09 blocks until all processors reach line 08 and logical processors $0, 1, \ldots, N-1$ reach line 12, i.e. complete the challenge execution. Then the $N$th logical processor continues by executing the challenge and encrypting its result with the public key of the authority. Finally the $N$th logical processor awaits completion of the challenge execution by all other logical processors. Forcing each logical processor to execute its own challenge, prevents an un-authenticated logical processor from copying the result of the challenge calculated by an authenticated logical processor.

We note that during the execution of the challenge by a logical processor, all other logical processors execute the MWAIT instruction. The MWAIT instruction does not affect the internal structures of the processor but prevent other core to affect the tested core.

## V. Performance Counter Chaining

Modern processors manufactured by Intel and AMD provide a facility to count occurrences of side-effect events, internal to the CPU circuitry, called performance events. The main goal behind this feature is to support CPU performance monitoring. Performance events are defined as internal CPU-circuitry state changes resulting from instruction execution, but not linked directly to the instruction results. For example: cache hit or cache miss events on specific cache memories, such as L1/L2/L3 or the translation lookaside buffer (TLB). The number of possible performance events greatly outnumber the available hardware counter circuits. Therefore, it is possible to dynamically link an available hardware counter (called a performance counter) to a specific performance event. Once linked, the performance counter counts the number of events that occurred.

In processors manufactured by Intel and AMD, performance counters are realized by a set of model-specific registers. Performance monitoring mechanisms were introduced with the Pentium processor and later evolved with the introduction of the P6 family, Pentium 4, core and all later processors. In general, some performance mechanisms are architectural. These performance counters are uniformly defined for all processors, while others are non-architectural, meaning they are specific to the micro-architecture and vary between the different processor families.

Most processor models are restricted to 2-4 individual performance counters, while the different Xeon-family processors are an exception in their capability to support 9-25 performance counters, depending on the exact model.

One of the challenge's goals is to determine if the remote machine is executing under emulation or not. Two factors are measured to determine this: the challenge result and the challenge's elapsed execution time. The underlying assumptions are that an emulating system shall evoke different performance events as compared to a non-emulating system. Therefore, in order to calculate a result that is correct for a non-emulating system, its performance-measured environment must also be emulated by an emulator attempting to masquerade itself. Taking the previous example, where TLB side-effects are measured, such an emulator would be required to maintain an emulated TLB in order to provide the same results a normally-running system would. A task that is not entirely impossible, but would surely introduce a detectable elapsed-execution time differentiation.

Even assuming that such a feat is possible with regard to one of the side-effect modules, referencing several modules in a single challenge would necessarily amplify the elapsed-execution time differences, since these emulations are mostly orthogonal.

Full emulation of Pentium processors accrue a speed toll estimated at 2-3 orders of magnitude without maintaining side-effects, which do not contribute directly to the software flow or results.

Adding side-effect emulation would increase the execution time by an additional factor. Hence, designing challenges that

utilizes a larger variety of performance measurements would not interfere with non-emulated system performance, while ensuring emulated-system differentiation.

A clear deficiency with respect to this is the low ration of available performance counters to possible performance events that can be measured. The novelty presented in this paper, designed to overcome this deficiency, is the use of "chained performance-counters". The idea is to monitor many side-effect inducing modules with a much smaller number of available performance counters, by shifting the counters from one module to the next according to a set of deterministic rules. All CPU components that generate side-effects are initialized to a known state before the challenge execution begins. When challenge execution flow reaches a determinable point, the contents of each side-effect inducing module is deterministic and repeatable regardless of our measurement i.e whether a performance counter was used to monitor its side-effects or not. It follows that a performance counter can be connected to the module to count new events. The new events will occur deterministically for the active challenge given the new determinable state.

As a result, monitoring performance events on multiple modules, using a single performance counter to measure the performance events of these modules, during several separate time intervals, will require a masquerading emulator to emulate all side-effect inducing modules to achieve the correct result.

## VI. Nondeterminism

Recent generations of modern processors have seen great advancement in pipeline optimizations, to gain significant improvements in throughput. In Intel and AMD processors, driving most of the worlds laptop, desktop and server systems, these include elaborate cache structures, branch-prediction circuits and prefetcher units. As a result, a lot is going on "behind the scenes" while the processor executes the main program thread. Statistically these predictive actions have a positive effect on performance — effectively increasing overall throughput. However, side-effect event counters are affected as well, leading to seemingly non-deterministic count results.

For example, consider counting L1 data-cache hits. When a load operation causes a new cache-line to be filled it is normally not counted as a hit. However, if that cache-line happened to be previously prefetched, the said load operation will be counted as a hit. Several prefetcher logic circuits exist that account for predictive loads to cache lines.

As discussed earlier, the challenge result calculation incorporates side-effect counter values while iteratively calculating a checksum. Since prefetchers are not directly controllable, indirect means need to be employed to defeat the prefetch logic and achieve deterministic challenge results.

## VII. Discussion

We have seen three advances in modern processors: virtualization extension, multiplicity of logical processors and a richer variety of performance events. Some advances improve the reliability of the authentication methods by increasing the resources required for a precise simulation. Other advances, on the other hand, allow un-authenticated remote machines to deceive the authority.

We have described how all the modern advances can be used to strengthen the authentication methods and mitigate deception attempts.

The main source of concern is the virtualization extension. While an active VMM can be easily detected on modern x86/x64 processors manufactured by Intel by issuing CPUID. The same task is much tougher on modern processors manufactured by AMD. A legitimate question to be asked is: whether, in theory, a slightly modified version of a virtualization extension can render an active VMM undetectable?

Consider an alternative implementation of the Intel virtualization extension in which the interception of the CPUID instruction can be disabled. Clearly, detection of an active virtual machine monitor is still possible and is similar to our proposal regarding AMD virtualization extension. Namely, we propose reading the register whose bits define whether a virtual machine monitor is currently active. On Intel, those bits are defined in control register 4 (CR4). Unfortunately, a virtual machine monitor can define the values of all control registers in the VMCS, rendering our detection method ineffective.

Another approach to detection of an active VMM is activation of another VMM. The second VMM is executed either directly by the hardware or with partial support of the currently active VMM. The latter form of execution, called nested virtualization, was implemented by some VMMs [22]. Since the hardware does not support nested virtualization, the VMM must react to certain virtualization extension instructions. By executing these instructions, we can differentiate between nested and regular virtualization.

We believe that it is possible, in theory, to implement a virtualization extension that will allow a virtual machine monitor to make itself undetectable. On the other hand hardware implemented nested virtualization. cannot be detected. The VMM can disable the interception of all events and schedule a delayed vm-exit. If the delay is longer than the execution time of the challenge then such a VMM cannot be detected during the challenge, but can establish full control over the processor during the delayed vm-exit.

## VIII. Conclusion

We have seen that the recent advances in instruction set architecture requires the authentication authority to introduce modifications to the currently existing attestation methods. Without accommodating the problems that arise from those advancements the attestation procedure will fail on modern hardware.

The problems attesting modern hosts vary from lack of predictability, caused by logical processor multiplicity, to unreliability, caused by virtualization extensions. While every problem requires a unique solution, we note that only paradigm shifting modifications of the instruction set architecture require redesigning the authentication challenges. We

have shownthat even those modifications do not prevent the establishment of authenticity of a remote machine.

## REFERENCES


[1] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251374

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095812

[3] 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.

[4] 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. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653711

[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. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2008.09.005

[6] 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. [Online]. Available: http://doi.acm.org/10.1145/1161289.1161306

[7] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1308172.1308237

[8] A. Seshadri, M. Luk, and A. Perrig, "Sake: Software attestation for key establishment in sensor networks," *Ad Hoc Netw.*, vol. 9, no. 6, pp. 1059–1067, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.adhoc.2010.08.011

[9] 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

[10] S. consumer electronics, "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

[11] 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

[12] Wikipedia, "An analysis of proposed attacks against genuinity tests," Tech. Rep., accessed on May 2015. [Online]. Available: http://en.wikipedia.org/wiki/Warden_(software)

[13] S. Pearson, *Trusted Computing Platforms: TCPA Technology in Context*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[14] 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. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1212691

[15] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251391

[16] 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. [Online]. Available: http://doi.acm.org/10.1145/1966913.1966957

[17] C. Tarnovsky, "Semiconductor security awareness today and yesterday," in *Blackhat 2010*, 2010. [Online]. Available: https://www.youtube.com/watch?v=WXX00tRKOlw

[18] ——, "Attacking tpm part two," in *Defcon 2012*, 2012. [Online]. Available: https://www.youtube.com/watch?v=Ed_9p7E4jIE

[19] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. van Doorn, "Remote detection of virtual machine monitors with fuzzy benchmarking," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 3, pp. 83–92, Apr. 2008. [Online]. Available: http://doi.acm.org/10.1145/1368506.1368518

[20] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3*, Intel Corporation, August 2007.

[21] "AMD64 architecture programmer's manual volume 2: System programming," http://support.amd.com/us/Processor_TechDocs/24593.pdf, 2010.

[22] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924973

# IV

## TIMING AND SIDE CHANNEL ATTACKS

by

Zaidenberg, N.J.; Resh, A. 2015

Cyber Security: Analytics, Technology and Automation, vol. 78, pp. 183-194

# Timing and Side Channel Attacks

Nezer Zaidenberg and Amit Resh

**Abstract** How would you know the US pentagon is planning an attack on Iraq? One possible plan is to infiltrate the pentagon using spies, flipping traitors etc. But this sounds like lots of work and it is a dangerous work. That is the direct approach. Another possible plan is to ask the pizza delivery guys in the area.

People planning an attack probably adds up to lots of people urgently trying to meet deadlines, staying late in the office and ordering pizza. So the pizza delivery guys know about a pending attack! The pizza delivery guys do not know the nature of the attack but they know "something is up" in the pentagon because for a few days people are staying late at the office and ordering pizza at irregular hours.

The pizza approach is the side-channel attack. This attack on the pentagon is not a direct channel attack. No spies were used. No attack on the pentagon defences. It is a side channel attack. Attack on the side effects of planning something. The people who plan need to work extra time and they also need to eat.

## 1. Introduction

In computing security there are numerous side channel attacks on side effects of verifying efficacy. These are not attacks that are designed on the primary line that was protected but on its side effects. A trivial such example is overcoming smartphone PIN protection on stolen smartphones. Normally there are 10,000 PIN combinations. However if the attacker can decipher the PIN digits by studying the smudge left on the device by the owner's fingers the problem can become a 1:24 problem, which is significantly easier. (Genkin et.al 2013)

This type of attack is very powerful. Schneier (2012) estimates that the NSA is able to break AES encryption using side-channel attack. Though no such method has been published.

N. Zaidenberg • A. Resh
Department of Mathematical Information Technology
University of Jyväskylä, Jyväskylä, Finland

# 2. Hypervisor Blue pills and Red pills

## *2.1. Subverting and Blue Pill concept*

A hypervisor is a type of software that allows running multiple operating systems on single hardware. The hypervisor treats a guest operating system in a similar fashion to the way an operating systems treats processes. The hypervisor manages the memory map for multiple operating systems in a similar fashion to the MMU of operating systems for processes. The hypervisor has a similar MMU for I/O devices, so one operating system is not effected by the I/O of another OS.

x86 allows multiple protection rings, the 2 most commonly used is Ring 0 for the Operating System (supervisor mode) and ring 3 for user code. Recently, x86 added ring -1 protection, a ring for the hypervisor, which adds instructions to create and trap OS operations. There are two different instruction sets for AMD and Intel hardware (AMD-v and VT-x instructions) but the two instruction sets are mostly polymorphic.

There are two possible hypervisors in x86 environments. Type 1(bare metal hypervisor) works directly over the hardware. The machine boots directly to the hypervisor. The hypervisor can then be used to boot other operating systems. We do not deal with type 1 hypervisor.

Type 2 hypervisor, which interests us, starts as a process (in ring 3) under an Operating system. (The operating system boots over the hardware normally). After the process starts it executes instructions that allow it to become a hypervisor. Thus the hypervisor gains more permissions then the OS that started it. After starting (as a ring 3 process) the hypervisor transfers the OS that started the process to a type of guest (getting into ring -1).

Joanna Rutkowska (2006) introduced the “bluepill” concept, a hypervisor which is barely noticeable. The hypervisor starts as a user process but gains complete control of the system (getting the user to run the process in the first place is a different problem). Rutkowska describes several such methods for example by hotel maids attacking hotel guest laptops etc.

## *2.2. Local Hypervisor Red Pills – Direct and Sub-channel attack*

How can the user know he is not running some hypervisor rootkit (such as the blue pill) on his computer? Direct attacks on the Bluepill involve actively trying to attack the hypervisor for effects that may not be hidden well or calling instructions that should be allowed if no hypervisor is installed but should be prevented if an hypervisor is already running.

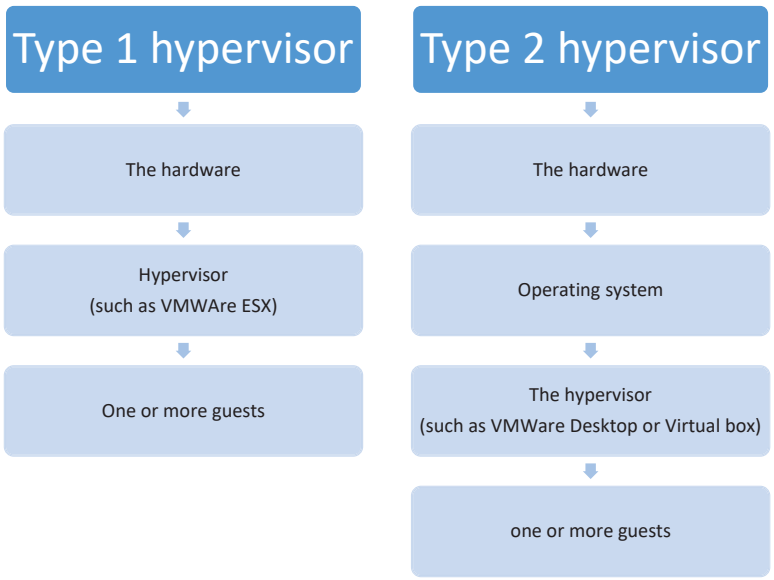


Fig 2.1 Type 1 and type 2 hypervisor

It has been shown that the Bluepill can be developed to hide masking detections attempts very well by direct means and even allowing infinitely recursive hypervisor calls. This led to sub-channel indirect attacks on the blue pill by measuring side effects of running the hypervisor. (Rutkowska 2006)

Perhaps attacks in this context are not clear. As it is the hacker that tries to install an invisible root kit the attacks in this context are by the lawful user who tries to detect and remove the “invisible” rootkit.

While a malicious hypervisor can mask itself from an attacker there are certain attacks that are bound to cause side effects. For example calling the CPUID assembler instruction should normally take about 200 cycles. But if a hypervisor is involved it will consume roughly 5000 cycles as a result of CPU context switches between guest and hypervisor mode. (CPUID always exits to the hypervisor)

Thus an attack on a naïve bluepill could be:

1. Measure CPU clock tick
2. Call CPUID
3. Measure CPU clock tick
4. If difference between 1 and 3 is greater the 400 cycles warn about blue pill

Off course as hypervisor based rootkits become more complex this attack can also be prevented. For example by trapping the call to get the CPU time.

There now exists a cat and mouse game in which more side effects can be measured (for example cache misses) and more attacks can be prevented by the hypervisor which cause more side effects that can also be attacked .(Rutkowska and Tereskin 2007)

## *2.3. Remote Hypervisor Red Pills*

The method described above demonstrates how the local computer can detect a malicious hypervisor rootkit as well as means that the hypervisor can use to hide itself better by catching more and more instructions.

The problem is that the hypervisor can control the guest, tricking it into believing the hypervisor does not exist. Also since red pill tools are offered to users, malicious hypervisor authors can reverse engineer any red pill and build a hypervisor that manages to mask itself against said red pill. However we can actually eliminate the cat and mouse game.

The problem lies with our check that is made locally using measuring tools that are all under the hypervisor control. However, the user can run his sanity test not locally, but using a cloud server.

The hypervisor doesn't control the cloud thus a 3rd party can efficiently detect if hypervisor is running. (It may be possible for the hypervisor to control the cloud response as it is caught by the OS but it is also possible for the cloud server to inform the user he is running a malicious rootkit using sub channel that the hypervisor does not control such as another computer)

Kennell and Jamieson (2003) have suggested a method for remote verification of the genuiness of a virtual machine[2] Kennel et al also include a mechanism to exchange an encrypted key with the authenticated host which was removed from our summary. (Shankar et. al 2004)

The jest of the kennel method can be summarized as following:

1. The cloud generates a random test. Tests are not identical and contain multiple steps.
   In each step side effects from the previous step are entered as input to the new step
2. The test executes in the inspected host
3. The inspected host sends a response.
4. The cloud serve verifies the response as well as the time it took to generate it
5. If the test was successful and within an allotted time the cloud server concludes that the host is genuine.

The steps in Kennell Memory Test include scans of the memory and instructions that run on the inspected OS. The side effects include TLB misses and other effects that are bound to produce different side effects if a malicious hypervisor is running.

Kennell et al (2003) argue that if the host is running some hypervisor there are bound to be different side effects. If the host is running an efficient emulator that also emulates all side effects the response will take too long to arrive.

---

[2] the papers uses the term "genuinity" however the correct English term is genuiness. We will use the correct English term in this chapter).

## 3. Invisible character differences

Let's assume we have some login screen (with username and password)

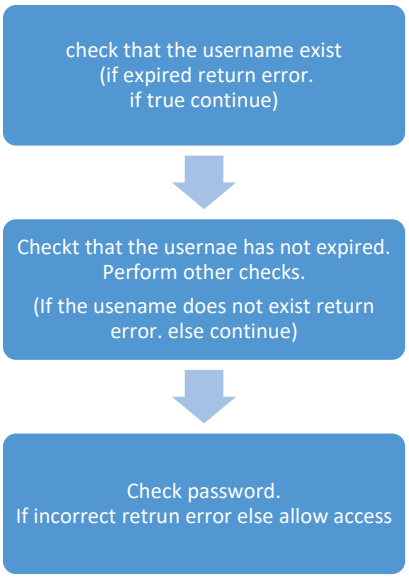


Fig 3 Username and password failure process

The login can fail for many reasons like:

- Username does not exist
- Username is expired
- Username is locked
- Password is incorrect

Of course for an attacker the different cases call for different behaviour. If the username is wrong there is no future with said username. If the username has expired the attacker may try again later. If the username is locked it is possible that the attacker activities have been detected. It is also possible to try later. Of course if the password is incorrect the attacker now has a correct username. The attacker can used the correct username for guessing the password (using dictionaries or brute force) or use the correct username for other attacks.

Assuming an attacker only wish to find out the correct username it would be critical to have all failure screens look identical. As by having a different failure screen for each case – guessing the correct username by brute force would be possible.

However, if the screens look identical, it is possible that several "invisible" differences exist. For example, if the communication is http communication transmitting a webpage, different web properties (setting cookies etc.) may exist for each of the cases)

# 4. Timing Attacks

Timing attacks occur when measuring the time it takes a system to respond. If the timing to receive a reply varies, not because of random variance in the delivery medium (such as the network time) but due to differences in responses and failure (for example different types of failure), an attacker can use the time variance to realize what type of failure has occurred. The information obtained using time measurements can later be used to attack the system.

## *4.1. GameCube DVD password attack*

The GameCube (Nintendo game console) was not supposed to play recorded DVD (pirated copies). The original DVD was released with copy protection system that was not trivial to replicate. Pirated copies were supposed to be detected by the copy protection system and be rejected by the system.

The DVD however had a programmable override (modchip) for the protection. The override may have been used by Nintendo in their system development. Had the password remained hidden we would never have heard about it.

However Nintendo checked the password using memcmp comparing byte after byte. The verification process ended when the first incorrect password byte was detected (returning failure) or when all bytes were compared successfully. Thus if the password got the first byte correctly the password check will be just slightly longer (checking two bytes instead of one) then if we got the first byte wrong (checking only one byte). Using this method one by one, all bytes that encompass the password can be revealed.

Thus the password was indeed leaked and 2$^{nd}$ generation GameCube modchips appeared (Domke 2004). Even though this method was well known, Nintendo had an identical problem with the Wii, which was released 5 years later. (Domke 2006)

Assume a webserver has a password protected section were username and password are required to login. When the user types his or her username and password, the algorithm from section 2 occurs. The System first checks for the username. If no such User exists (or if the user has expired) the system returns with an error. If the user is OK then the system now verifies the password. If the password is not OK then an error message will appear.

Assuming the answer is immediate, by timing the response times an attacker can use this timing attack to reveal correct usernames. Furthermore, even if the response occurs over some network which adds random delay (but similar random delay to both correct username and incorrect username – an attacker may still be able to guess the password. (Domke 2004)

Adding short random delays to password checking does not prevent timing attacks. As long as the delays are not significant it can be shown that an attacker can still distinguish between two classes such as incorrect username and correct username but incorrect password. (Domke 2004)

# 5. AES Side-Channel Attacks

## *5.1. AES Background*

AES (Advanced Encryption Standard) is the standard electronic-data symmetric-key encryption algorithm, specified by the NIST (US National Institute of Standards & Technology) since 2001 (AES 2001). It was labelled by the NIST as FIPS publication 197 and is based on the Rijndael algorithm, proposed by two cryptographers from Belgium: Joan Daemen and Vincent Rijmen (2013). In addition to being adopted by the US Government it is also used for data transfer and communication worldwide as a successor to the Triple-DES, DES and RCx cryptographic algorithms. This finds use in many implementations, most notably the SSL3/TLS protocol, as well as disk encryption and authentication.

AES is used to encrypt and decrypt fixed blocks of 128 bits (16 bytes). The cryptographic algorithm uses three possible key sizes: 128 bits, 192 bits or 256 bits. AES encryption and decryption is performed in several iterations, called “Rounds”. During each round, 4 steps (only 3 steps in the last Round) are performed on the intermediate data block to progress the encryption from a 16-byte plaintext (*ptext*) to a ciphertext (*ctext*). During decryption, similar steps are performed to achieve the opposite: decrypting the *ctext* to restore the *ptext*. The number of rounds used depends on the key size: 10, 12 and 14 rounds are used for key sizes of: 128, 192 and 256 bytes.

The encryption process can be summarized as follows:

A. Key expansion: The original 128-bit key is expanded to 10, 12 or 14 Round-keys. The data block in each round is combined with the Round-key corresponding to that Round.
B. Initial round: Each *ptext* byte is combined with the original key
C. Rounds (all but last): activate 4 transformations on the data buffer: SubBytes ; ShiftRows ; MixColumns ; Combine with round-key
D. Last round: activate 3 transformations: SubBytes ; ShiftRows ; Combine with round-key

The decryption procedure is similar, using the same original key but inverse-transformations.

## *5.2. AES Software Implementation*

Software implementations of AES normally make use of lookup tables in favour of performance and efficiency. The lookup tables are used to quickly determine the transformation results, which are activated in the AES rounds. While theoretically it is possible to calculate the transformations without resorting to lookup tables, using

only arithmetic, logic and Boolean operations, this carries a significant performance toll that is naturally avoided.

Accessing lookup-tables that have a substantial size (as is the case for AES encryption/decryption) interacts with the underlying architecture's cache mechanism. As we shall see below, this provides an opening for a side-channel attack crafted to reveal the key.

### *5.3. Cache Memory*

Modern CPU systems have several levels of storage. They differ generally by a capacity vs. access-speed trade-off.

Cache memory is used to buffer memory contents obtained during a memory cycle, so it can be provided much faster when it is needed in a following memory cycle. Cache operations are generally transparent to software and dedicated hardware is used to manage the buffering and utilization cycles of cache.

Cache memory is subdivided into cache-lines. When a memory element is stored in cache, an entire cache-line (which contains that memory element) is stored in cache. The cache circuitry kicks in at every physical memory access.

Cache operation during a memory read cycle is best explained with an example:

> When a memory location is read-in by the processor, the cache is first inspected to determine if the required value can be obtained from the cache. If it can, this event is called a "cache hit" and the value is provided to the CPU directly from the cache. Such a memory cycle is significantly faster than retrieving the value from main memory. If the required value is not in cache, it is called a "cache miss" and the value must be retrieved from main memory. In this case the value is both provided to the CPU and stored in cache. The cache-miss event causes an entire cache-line to be retrieved from memory and stored to cache, called a cache-line fill. The next access to any memory location within that cache-line will be a cache-hit.

When a cache-miss occurs and, as explained above, a cache-line is retrieved and stored in cache - some previously stored cache-line needs to be evicted from the cache to make room for the new one. Usually cache-lines are evicted according to an LRU (Least-Recently-Used) algorithm.

As mentioned above, memory elements are stored to the cache in integral quantities of cache-lines. The address of a memory element is subdivided into 3 fields. The *index* field determines which cache-slot will be used. Note that all address locations that contain the same *index* value, share the same cache-slot. Each cache-slot also stores the *tag*, in addition to the cache-line content. The tag is used to define the exact memory location of the cache-line that is currently in the cache-slot. The LSBits are the *offset* field and define the offset of the memory element within the cache-line.

This mapping is referred to as the 'Cache Association'. See figure 5.3.1 that depicts the cache structure and Association. When a cache-line fill operation occurs, a cache-slot will first be evicted (written back to memory) and then filled with the new cache-line contents.

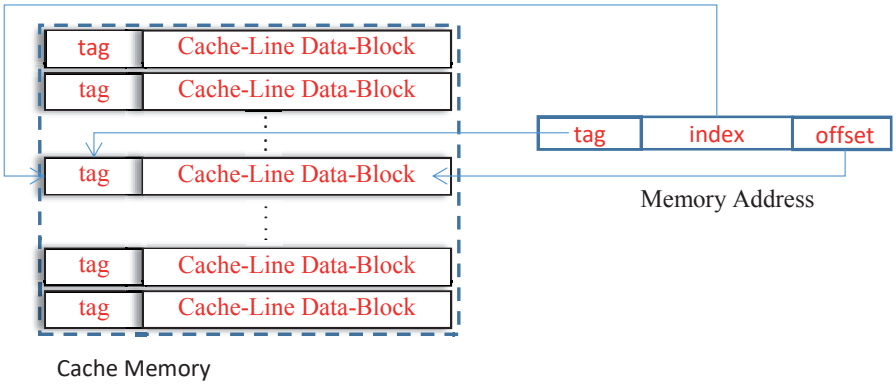


Fig 5.3.1 Cache Structure and Memory Association

The figure 5.3.1 above depicts the mapping between a memory address and a specific cache-slot. This is called a 1-way association. Modern cache designs boost performance by increasing the number of available cache-slots for each *index* value. When more slots exist, a cache-line is not necessarily evicted when a new memory location with the same index needs to be stored to cache.

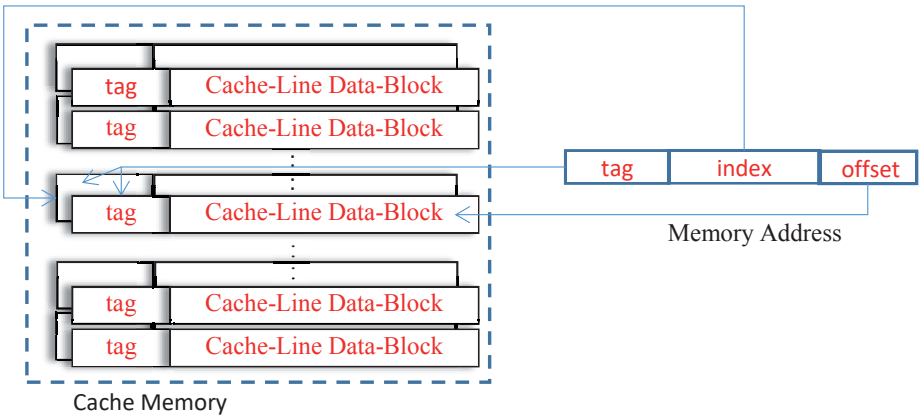


Fig 5.3.2 2-Way Set Associative Cache

For example, in a 2-way association, each cache-line index has 2 separate available slots. When a new cache-line is filled, only one slot must be evicted to make room for the new one. See figure 5.3.2Fig. An LRU (Least-Recently-Used) algorithm is usually used to decide which slot should be evicted. This architecture boosts performance, since recently used memory values have a better chance of remaining in cache and therefore increasing the cache-hit ratio. It is not uncommon

in modern CPU architectures to have 4-way and 8-way set associative cache designs.

### *5.4. Side Channel Attacks on AES*

Several attack strategies have been developed against the AES algorithm. Some attempt to acquire the memory contents of the AES application using different types of stealthy techniques, such as DMA attacks, Cold-Boot attacks or use of Firewire, PCI, etc. Once captured, memory contents can be analysed to retrieve the key used.

Another category of methods resort to an indirect strategy, which does not attempt to access the AES application directly, but takes advantage of side-effects that occur in the computer system as a result of the execution of the AES application and which can eventually lead to revealing the key. As explained above, these belong to the "Side-Channel" attacks category.

Cache-timing side-channel attacks are based on the fact that the processor accesses a cached memory element (*cache-hit*) at a significantly faster cycle time than that of a non-cached one (*cache-miss*). Different applications on the same system are protected from each other with Virtual memory; however the same underlying cache structure services all processes that run in parallel on the same CPU. Multiprocessing is supported by virtually all Operating-systems in use today. Consequently, if one process affects the cache subsystem, another parallel process can measure that affect even if it is restricted to accessing its own, private, address space. The timing differences between a cache-hit and a cache-miss are a factor of x10 – x20. This leaves ample leeway for one process, running along-side another process on the same CPU to accurately measure those affects.

Recall that software driven AES applications make extensive use of lookup-tables. When a lookup-table entry is referenced it will be retrieved from the cache in the event of a cache-hit. Otherwise, in the event of a cache-miss it must be retrieved from main memory with a time penalty. An attacker routine, which runs in parallel to the AES process can measure the time of the encryption or decryption and compare the measurements when a specific lookup-table entry exists and then does not exist in the cache. One way for the attacking process to achieve this is to evict the specific lookup-entry from cache. To do that, it only needs to reference memory elements from its own memory space, which has the same *index* as that of the lookup-table entry. Doing so for enough memory elements (at least the cache association ways) guarantees that the lookup-table entry is evicted from cache. For example, in a 4-way associative cache, 4 references to memory elements with the same *index* will evict the lookup-table entry. Following this, 2 consecutive AES decryptions are triggered and timed. If the first time measurement is longer than the second, it can be concluded that the specific lookup-table entry in question was referenced. Otherwise, the converse is true. Since lookup-table references occur as a function of the key value, repeating this process for different lookup-table entries can be used to reveal the key value. Additional details can be found in the work of D. Osvic, A.Shamir and E.Tromer (2005).

Alternative methods may be employed by the attacking process to employ the same principles. For example, the entire cache may be evicted (this is usually a single instruction), followed by triggering of an AES decryption. The state of the cache would then reflect all the lookup-table entries that have been referenced. Now the attacking process can time references to memory elements in its own memory space, which have the same *index* value as a specific lookup-table entry. If the measured time corresponds to a cache-hit, it can be assumed that the lookup-table entry was referenced during the AES process. If the measured time corresponds to a cache-miss the converse can be concluded.

AES software implementation may be written to obscure the use of lookup-tables in such a way that it becomes impossible to relate its use to key values. Alternatively the implementation can avoid use of lookup-tables altogether. The down side of these methods is the performance penalty. Use of lookup-tables is the fastest (albeit vulnerable) implementation.

In 2010 Intel introduced a new instruction set in the Westmere processor family to perform AES calculations in hardware. This instruction set is dubbed AES-NI. The instruction set consists of 6 instruction op-codes: 2 for key expansion and 4 for encryption and decryption. By using these instructions the entire AES process is carried out in hardware in fixed, data-independent, timing. As a result, cache-timing attacks become completely useless. (Gueron 2012)

# 6. Power based attacks

RSA is a common cryptographic method that relies on mathematical operations (mainly multiplications and divisions). However multiplication has different power requirements for bits containing 0 (hereby “0-bits”) compared to bits containing 1 (hereby “1-bits”). Due to the arithmetic nature of multiplication, multiplication involving “0-bits” is equivalent to NOP. Multiplication of “1-bits” on the other hand consumes more power for CPU operations.

Assuming the attacker has access to the platform were RSA or similar protection algorithm is running, during validation of the correct key. It has been shown that using the power consumption of the platform the attacker can detect the “1-bits” in the key thus breaking the encryption. (AES 2001)

Protection against power based attacks involves doing similar operations for “0-bits” and “1-bits” by doing random computations for “0-bits”. This method increases the computation time of RSA and similar algorithms as it adds random computations. It is also still vulnerable to attacks as computation is not 100% identical, however it has been shown that by adding random CPU work to “0-bits” the power consumption gap between “0-bits” and “1-bits” can be eliminated.

Closely related to Power-analysis side-channel attacks, are Acoustic side-channel attacks. Computer systems emit (ultrasonic) acoustic sound as a result of current surges through electronic components, such as capacitors and coils. Monitoring and

analysing these sounds can reveal the underlying current consumption graph of the computer system. Therefore, a cryptographic system may be attacked in much the same way as one whose power usage is monitored. For implementation details see the work of Genkin, Shamir & Tromer (2013). Acoustic monitoring has a distinct advantage in that a physical connection is not required, as measurements can be achieved solely by using a sensitive microphone.

Countermeasures that defeat these attacks may be to generate a random variety of sounds in the same spectrum, while computing the critical cryptographic algorithms. White-noise can also be used to acoustically drown the side-channel emissions.

## References

Advanced Encryption Standard (AES) (2001), United States National Institute of Standards and Technology (NIST), Federal Information Processing Standards Publication 197, 2001

Daemen, Joan and Rijmen, Vincent (2013), AES Proposal: Rijndael. s.l, National Institute of Standards and Technology, 2013. p.1

Domke Felix (2004), Console hacking 2004, CCC 2004

Domke Felix (2006), Console Hacking 2006, CCC, November, 16th 2006

Genkin, Daniel; Shamir, Adi; Tromer, Eran (2013), RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. tau.ac.il, 2013

Gueron, Shay (2012), Intel® Advanced Encryption Standard (AES) Instructions Set - Rev 3.01. s.l.

Kennell Rick & Jamieson Leah H. (2003), Establishing the Genuinity of Remote Computer Systems, Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4–8, 2003

Osvik Dag Arne; Shamir Adi; Tromer Eran (2005), Cache Attacks and Countermeasures: the Case of AES

Rutkowska Joanna (2006), Introducing Blue Pill, http://theinvisiblethings.blogspot.fi/2006/06/introducing-blue-pill.html

Rutkowska Joanna and Tereskin Alexander (2007), IsGameOver() Anyone?, Invisible Things Lab, 5.8.2007

Schneier Bruce (2012), Can the NSA Break AES?, Schneier on Security blog, www.schneier.com/blog/archives/2012/03/can_the_nsa_bre.html

Shankar Umesh, Chew Monica, Tygar J. D. (2004), Side effects are not sufficient to authenticate software, Report No. UCB/CSD-04-1363, September 2004, University of California, Berkeley, California

# V

## TRUSTED COMPUTING AND DRM

by

Zaidenberg, N.J.; Neittaanmäki, P.; Kiperberg, M.; Resh, A. 2015

Cyber Security: Analytics, Technology and Automation, vol. 78, pp. 205-212

# Trusted Computing and DRM

**Nezer Zaidenberg, Pekka Neittaanmäki, Michael Kiperberg, Amit Resh**

Department of Information Technology, University of Jyväskylä, Jyväskylä, Finland

## 1 Introduction

Trusted Computing is a special branch of computer security. One branch of computer security involves protection of systems against external attacks. In that branch we include all methods that are used by system owners against external attackers, for example Firewalls, IDS, IPS etc. In all those cases the system owner installs software that uses its own means to determine if a remote user is malicious and terminates the attack. (Such means can be very simple such as detecting signatures of attacks or very complex such as machine learning and detecting anomalies in the usage pattern of the remote user).

Another branch of attacks requires protection by the system owner against internal users.

Such attacks include prevention of users to read each other's data, use more than their allotted share of resources etc. To some extent anti-virus/anti-spam software is also included here. All password protection and used management software are included in this branch.

The third branch, *Trusted Computing*, involves the verification of a remote host that the user machine will behave in a certain predictable way, i.e. protection against the current owner of the machine. The most common example for this kind of requirement is distribution of digital media. Digital media is distributed in some conditional access mode (rented, pay per view, sold for personal use, etc.). Obtaining digital media usually does not entitle the user to unlimited rights. The user usually may not redistribute or edit the digital media and may not even be allowed to consume it himself after a certain date. (Media rentals, pay per view) However, as the user is consuming media on his private machine. How can the media provider assure himself that a malicious user does not tamper with the machine so that contents are not replicated? The problem of security against the owner of the machine is the problem region of Trusted Computing. In trusted computing as opposed to other branches of security the "attacker" is not limited to some attack surface that was exposed to him but can also use a soldering iron to tap into busses, replace chips and other system parts etc.

Trusted computing also includes other protection tools against the current owner (or possessor of the machine if not the legal owner). For example protection of

sensitive data or disk encryption solutions for laptops and mobile phones that can potentially be stolen.

Trusted computing can also be used on the cloud to ensure that the host does not inspect a cloud server and the software running on the server is not stolen. Latest trusted computing technology involves means to ensure commands are sane and are not malicious, for example in computers on cars and avionics. In this chapter we will review DRM and Trusted computing solutions from multiple sources.

## 2 Ethics – Trusted or Treacherous computing

Users don't like trusted computing.

First and foremost, the concept of conditional access leads to numerous digital rights debates. For example, if I legally purchased contents, shouldn't I be allowed to make backups of said contents? Especially as no media vendors are currently proposing to offer free (or even cheap) replacements of corrupted media contents! However, if we allow media to be replicated then how can we disallow illegal copies? What is stopping the user from redistributing copies or "backups"? How can we distinguish legal use of copies (backups) and illegal copies of the same content?

Secondary, as many trusted computing devices requires the user to actively install something on his machine (a TPM chip, EFI firmware, etc.). And the said hardware component does not contribute to the end-user system features at all (if anything trusted computing only limits the user). Why would the user willingly spend money and install some piece of hardware in his computer that only serves to limit what she can and cannot do?

All these reasons have lead Richard Stallman to call trusted computing treacherous computing and numerous hackers to try attacks on TPM chips and trusted platforms.

As of writing this chapter there is no clear cut winner in this technology battle. On the one hand there is still no massive install base for trusted computing solutions and on the other hand the trusted computing group is still alive and still releasing new trusted platform modules and specs.

# 3 The Trusted Processing Module by TCG

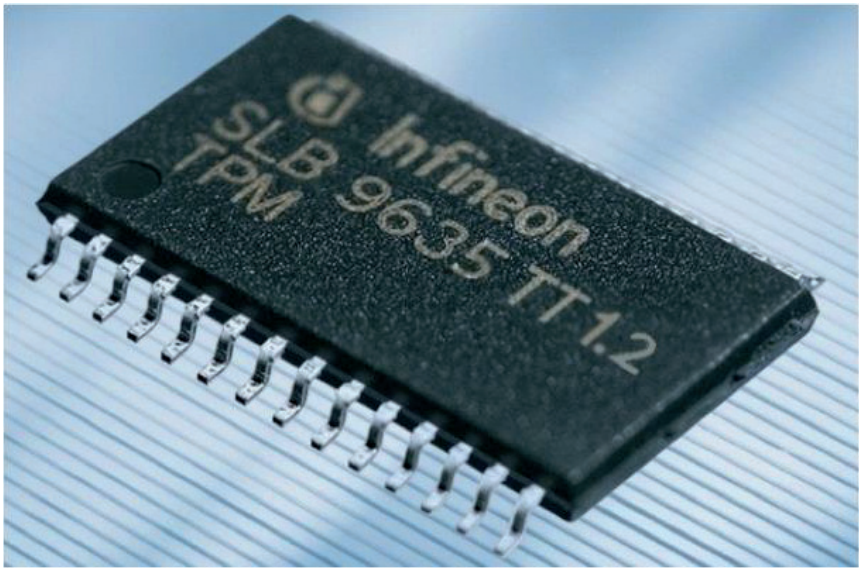


Figure 1: Trusted platform module

The trusted platform module, as demonstrated in Fig. 1 is a separate computer chip that is added to the computer motherboard and is frequently connected to the CPU using the LPC (low pin count) bus. The TPM is a cryptographic co-processor that is able to perform several cryptographic functions as well as generate and store keys.

The TPM can also verify the hardware and software that the system runs on and attest for the system's sanity.

When a remote host is querying the system for sanity it can use the TPM to verify that the software that it runs on was not tampered. TPM supports two attestation methods: Remote Attestation and Direct Anonymous Attestation.

## *3.1 Remote Attestation*

The attestation solution proposed by the TCG (TPM specification v1.1) requires a trusted third-party, namely a *privacy certificate authority* (privacy CA). Each TPM has an RSA key pair called an Endorsement Key (EK), embedded inside the TPM (that the user cannot access – at least not easily.)

In order to attest itself, the TPM generates a new RSA key pair (Attestation Identity Key or AIK).

Remote Attestation is a typical trusted 3rd party process. Assuming Alice wants to attest Bob and Bob wants to be recognized by Alice but neither wants to reveal his private identification code to each other, remote attestation is suggested. It requires a trusted party that both can trust.

Bob attests himself by signing the public AIK using the EK, to the trusted 3rd party. The trusted 3rd party then verifies Bob by using Bob's public EK. Of course the CA may and should blacklist TPMs if it receives too many requests using the same key simultaneously. Alice can later verify with the CA Bob has is indeed attested.

### *3.2 Direct Anonymous Attestation*

The Direct Anonymous Attestation (hereby DAA) protocol was only added to the TPM standard in version 1.2. DAA is based on three entities and two steps. The entities are the TPM platform, the DAA issuer and the DAA verifier. The issuer is charged to verify the TPM platform during the Join step and to issue DAA credentials to the platform. The platform uses the DAA credentials with the verifier during the Sign step. The verifier can verify the credentials without attempting to violate the platform's privacy (zero knowledge proof [5, 6]). The protocol also supports a blacklisting capability, so that verifiers can identify attestations from TPMs that have been compromised.

DAA allows differing levels of privacy. Using DAA Interactions is always anonymous, but the user/verifier may negotiate as to whether the verifier is able to link transactions (with the same user but not a specific user). Verifying transactions would allow persistent data to be saved over sessions and would allow profiling and tracing multiple logins.

## 4 Intel TXT instructions

Intel TXT technology defines unique extensions for the CPU instruction set to allow trusted execution [2]. Using intel TXT, one can attest the hardware, OS and software currently running and ensure a stable (as opposed to tampered) system state. Intel TXT uses the TPM for measurements and cryptographic functions to attest to a 3rd party and ensure that system software or the OS that is currently running is indeed trustworthy and non-tampered with.

The PCR registers on the TPM contain measurements and SHA-1 hashes of various system stages and code and by checking and verifying these measurements the system can be trusted to boot a non-tampered software.

# 5 AMD/ARM Trustzone

AMD/ARM Trustzone is the ARM/AMD implementation of trusted computing. It is roughly corresponding to Intel TXT. The Trustzone implementation is used by both AMD and ARM. Trustzone allows signed secure OSs to be loaded, for example, by using AMD/ARM SVM.

# 6 Other architectures for "Trusted computing"

These architectures provide means to prevent replication of data and thus introduce trust on various systems. We focused mainly on Video content delivery in this chapter. Different systems for preventing homebrew and pirated software on game consoles (which is another form of trusted computing) are covered in Chapter 3.

## *6.1 HDMI and HDCP and its predecessors*

The Video industry has always been interested in mixed goals:

1. It searched for ways to deliver high quality video to the user's home. Generating a new revenue stream from videos that no longer appeared in cinemas (Video/DVD rentals).
2. It searched for ways to prevent the user from obtaining permanent access to the video equipment she rented by making illegal copies.

To some extent the battle was a lost cause to begin with because the user could always point a standard camera to the screen and just record using the camera (or create low quality copies using older, already broken technology). However, the industry was interested in preventing the user from making high quality copies (for example, digital quality copies in the case of HDMI).

This approach led to several technologies whose purpose was to circumvent the user's ability to create illegal copies

## *6.2 Macrovision*

Old VHS video devices had a macrovision device that prevented direct creation of copies of VHS media by connecting two video devices to each other.

The Macrovision devices modified the output stream in a way that was unnoticeable to users but prevented VHS devices to create VHS cassettes copies by daisy chaining devices.

## *6.3 CSS and DeCSS and improvements*

CSS or Content Scrambling System is an encryption system that is used on all major DVDs.

CSS was 40 bit encryption system.

The use of CSS was supposed to make it impossible to copy video content directly from DVD to a video. This was done as the encryption keys were kept in unreadable (by data DVD players) location.

CSS also allowed for DVD regions, Macrovision etc.

DVD CSS was broken at 1999, about 3 years after it was introduced with the introduction of DeCSS software. An inherent bug was used to reduce the keys from 40bit to only 16bit long and most players were able to break this encryption in less than 1 minute by brute force.

Two of DeCSS authors remain unknown even today. The 3rd was a Norwegian teenager: Jon Lech Johansen. Mr. Johansen was brought to trial and acquitted by the Norwegian court. The prosecution appealed and Mr. Johansen was acquitted for the second time.

When DVD was superseded by Blu-Ray and HD-DVD CSS was replaced with the AACS (Advanced access contents system, which was broken using leaked keys).

## *6.4 HDMI and HDCP*

HDMI or High Definition Media Interface is a high quality media interface allowing high quality media transfer to monitors and screens. HDMI raised the problem of creating exact or near exact high quality replicas of video content.

To avoid copying the contents, HDCP will encrypt the content travelling between two end points of HDMI and will only provide contents to devices with trusted keys. These keys can later be revoked if they are stolen.

By 2010 the master key for HDCP had been leaked, rendering all revocation list useless.

It is possible that the revocation key was used too many times and provided sufficient data that made breaking the key easier.

# 7 Other uses for trusted computing

Several attacks on the user can be done after the attacker has obtained full or even partial control on the end user device. For example the attacker may be interested in the contents of the user hard drive after obtaining control of the user laptop (for example, by stealing it)

Such trusted computing content protection methods involve the usage of protected (encrypted) storage were the keys are saved on the TPM.

## *7.1 Microsoft Bitlocker and similar products*

Microsoft Bitlocker is a full disk encryption solution that can be used on computers (especially laptops) to ensure that the disk contents are unreadable to an attacker, even if the computer/laptop was stolen. The complete disk is encrypted and the key to decipher the disk content is unreadable and saved on the TPM.

## *7.2 Protection on Mobile phone data*

Mobile phones contain private data that can be exposed if the phone is lost or stolen.

Numerous technologies have been generated by various sources from using TPM and encryption on the device to a more biometric approach.

Examples include apple usage of fingerprint reading devices on the iPhone device that are required to unlock a stolen phone.

Other technologies include a kill code that is used to wipe the device and prevent it from connecting to the network ever again.

# 8 Attacks on trusted computing

## *8.1 Reset Attacks on the TPM chip*

The TPM is often connected to the LPC (low pin count) bus. A legacy slows bus that exist on virtually all PCs. Attacks on this exist for over 10 years. One of the first cases of attacks on this bus occured on the first XBox.[9] By connecting

and eavesdropping to the LPC bug several hackers have been able to intercept and reset the TPM[8].

### *8.2 Attacks on the implementation of TPM*

In 2010 [3] and later in 2012 [4] Chris Tarnovsky demonstrated physical attacks against TPM chips by Infineon and ST microelectronic. Tarnovsky attacked the TPMs by eliminating parts of the TPM chips thermal casing and attacking (i.e. connecting external devices) to the chips itself.

Tarnovsky demonstrated that ST and Infineon chips are made of older processors chips from their past. He demonstrated that by physically attacking the chips itself he could expose and modify content on the TPM chip itself.

Tarnovsky's methods require a special lab, chemicals and equipment which may not be in every hacker's reach. But it is definitely not beyond the reach of professional attackers and hackers.

### *8.3 Other attacks on trusted computing*

One of the features of the Intel CPU is SMM or software maintenance mode. SMM is used to allow updating CPU code (microcode). SMM code is executed at higher permissions then user, kernel or hypervisor code on the Intel platform. Therefore, SMM is considered to run at permission ring -2 (if Ring 3 is userspace code, ring 0 is kernel code and ring -1 is hypervisor)

Rafal Wojtczuk and Joanna Rutkowska have demonstrated breaking TXT limitations using SMM [1]. These attacks may have been Intel's main reason for devising the SGX extension.

These attacks are possible because TXT protection blocks execution and permission in rings 3 (user space), 0 (kernel) and -1 (hypervisor) but TXT memory defense is still vulnerable to attacks on Ring -2 using SMM permission level which does not require any special permissions and can be used even after the OS has been attested by the TPM.

## 9 Beyond Trust – SGX

SGX or Software Guard Extension is an innovative technology from Intel that will be implemented in future chips. SGX provides a solution to the trusted computing problem on Intel platforms [2]. SGX technology allows creating an execution container for each process in which the process memory is contained. This

approach is similar to the approach taken by Qubes OS development to create separation using hypervisor code between applications so different applications are running on different virtual OSs [6] and by Trusted computing software such as TrulyProtect, which keeps secrets in the hypervisor layer [7]. At the time of writing this chapter SGX is not available with any Intel CPU on the market (thus there are no known attacks on SGX).

## Bibliography

[1] Rafal Wojtczuk and Joanna Rutkowska. Blackhat DC 2009 “Attacking Intel® Trusted Execution Technology”
[2] Intel Trusted execution Technology - whitepaper hardware based technology for advanced server protection http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html
[3] Chris Tarnovsky Defcon 2012 “DEF CON 20 - Attacking TPM Part 2 - Chris Tarnovsky”
[4] Chris Tarnovsky. Blackhat DC 2010 “hacking the smartcard chip”
[5] Quisquater, Jean-Jacques; Guillou, Louis C.; Berson, Thomas A. (1990). "How to Explain Zero-Knowledge Protocols to Your Children". Advances in Cryptology – CRYPTO '89: Proceedings 435: 628–631.
[6] Blum, Manuel; Feldman, Paul; Micali, Silvio (1988). "Non-Interactive Zero-Knowledge and Its Applications". Proceedings of the twentieth annual ACM symposium on Theory of computing (STOC 1988): 103–112
[7] Nezer Zaidenberg ECIW 2013 “TrulyProtect 2.0 and attacks on TrulyProtect 1.0”
[8] TPM Reset Attack Evan Sparks http://www.cs.dartmouth.edu/~pkilab/sparks/
[9] Michael Stiel “17 mistakes microsoft made with the xbox security systems”

# VI

## CAN KEYS BE HIDDEN INSIDE THE CPU ON MODERN WINDOWS HOST

by

Resh, A.; Zaidenberg, N.J. 2013

ECIW 12th European Conference on Information Warfare and Security,
Jyväskylä

# Can keys be hidden inside the CPU on modern Windows host

Amit Resh
Nezer Zaidenberg
University of Jyväskylä, Jyväskylä, Finland
amit@truly-protect.com
nezer@truly-protect.com

## Abstract

The “Truly-Protect” trusted computing environment by Averbuch et al relies on encryption keys being hidden from external software and crackers. “Truly-Protect” saves the keys in internal registers inside the CPU. Such external keys should not be accessible by any software that runs on the machine prior to “Truly-Protect” validation or even after “Truly-Protect” validation. The assumption is that the hackers cannot reverse engineer the CPU and discover the content of these registers. But is it really possible to hide keys in such places?

Internal CPU memory is indeed not available for user processes. However, the CPU memory and registers are accessible from the running operating system kernel. Truly protect uses a validation protocol that also verifies the Operating system kernel does not include malicious additions. These tests should ensure a cracker has not modified the OS. But Modern Windows operating system support loading new kernel code segments (drivers) even during the operating system runtime. Can we prevent modifying the kernel (loading drivers) after “Truly-protect” has verified the kernel?

In this work we examine modern Intel CPUs available on desktop PCs and the latest releases of Microsoft Windows (windows 7,8) for existence of good hiding places for the encryption keys.

## 1. Introduction

Contemporary digital rights security systems rely mainly on methods of obfuscation or use of plug-in HW devices, such as dongles. Use of HW dongles has been critiqued heavily by users, as being cumbersome and generally inconvenient. Obfuscation methods, by which software protection is realized by introducing code-clutter to conceal the protection mechanism is largely losing the battle to crackers, who on average can break these protection schemes within weeks. A new approach, described by Averbuch et al [1] suggests a software-only solution, named Truly-Protect, based on encryption and just-in-time decryption of protected software. According to this approach, the protected software shall be stored in computer memory exclusively in its encrypted form. Decryption shall occur "inside the CPU", on-the-fly, as it is being consumed. The decrypted form shall not be stored back into memory. In fact, it shall never leave the confines of the CPU domain. See shaded area in Figure 1.
Software protection based mainly on obfuscation still allows crackers to trace and reverse-engineer the protected software, thereby opening the door to obtaining an unprotected copy. However, by keeping the decryption process and its keys, as well as the decrypted results inside the CPU domain assures that the software remains protected -- unbreakable by any of the currently know cracking techniques.

According to Truly-Protect [1] the following procedure is used in order to successfully execute protected software on a target computer:

- The target computer communicates to a remote authentication server and transmits proof of eligibility to execute protected software.
- The remote server authenticates the target computer by employing a modified Kennel & Jamieson [2] procedure. The purpose of this step is to validate the target computer as a real (non-Virtual) machine running a recognized O/S. A side-effect of the validation procedure is exchange of key material.
- The server protects the software by encrypting it using the key material exchanged with the target during the validation procedure.

- The protected (encrypted) software is downloaded to the target's memory and spawned for execution.
- Protected software executes on the target computer using JIT, on-the-fly, decryption: Encrypted instruction code is loaded from memory into the CPU, where it is decoded, executed and then disposed. Decryption keys or decrypted instruction codes never leave the CPU domain.

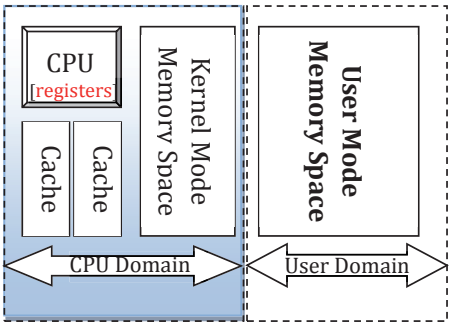


**Figure 1: CPU & Memory Structure**

## 2. Problem Definition

A full description of the validation and key-material exchange procedures, detailed in the procedure above, is beyond the scope of this discussion. We will proceed with the assumption that the validation procedure establishes the following:

- The target computer is a real (non-VM) system
- The target is running a recognized O/S that does not include potentially malicious components
- The key material, required for decrypting the protected software, is generated by the validation procedure and stored in the CPU domain

The Truly-Protect scheme is based on maintaining keys and carrying out the protected software decryption exclusively in the CPU domain. This implies that the decryption code runs in Kernel-mode (privilege level 0) on a protected O/S, such as Windows or Linux. This further implies that decryption must either be an integral part of the O/S or a Driver that is loaded into the O/S and operates in Kernel-mode. While this restriction in itself is a complication, it is a blessing in terms of software protection, since it establishes a basis upon which the Truly-Protect goals can be realized.

The protected software is assumed to execute in user-mode. However, according to Truly-Protect, decrypted code cannot exist outside of the CPU domain. This restriction implies that decrypted code must either:

- remain in the CPU register file for the duration of its execution, or
- be latched in cache while the cache method for that space is set to Write-Back

In the former case individual instructions must be decrypted and executed by a VM, while in the latter case, large blocks of code (for example, entire functions) may be decrypted and executed natively, directly from cache. These ideas have been described in detail in [1].

The most crucial aspect of Truly-Protect and its "soft-belly" is the decryption-key location. Once generated by the validation procedure, it must be locked in the CPU domain, such that it cannot be accessed under any condition, except, of course, to carry out the JIT decryption. As mentioned above, locating information in the CPU domain restricts its access to the O/S Kernel or driver modules executing in Kernel-mode. Therefore, storing the keys anywhere in the CPU domain will keep it safe from User-mode applications. We assume that during the validation procedure, when the key is initially generated, the CPU domain is clean of

malicious code. However, how can Truly-Protect guarantee that malicious Kernel-mode drivers are not loaded thereafter, gain access to the CPU domain and get hold of the key -- thereby using it to completely decrypt the protected software?

Several approaches may be employed to accommodate:

- Lock the key in a memory region that can be accessed only by the decryption engine
- Prevent driver plug-ins to the O/S after the validation procedure has successfully completed.
- Allow Kernel related changes or driver additions – but in the event that these occur – the key must be obliterated

## 3. Discussion of Alternative Solutions

**Cache**

Cache memory is one good storage place, in which to conceal key information deep within the confines of the CPU domain. Most modern age computer systems contain one or more cache units for Instruction, Data or Unified caching. For example, the Intel Pentium processors contain 3 levels of cache units: L1 (Instruction and Data), L2 (Unified) and L3 (Unified).

Cache memory cannot be read or written directly by software (User or Kernel mode) as internal cache mechanisms maintain correspondence between cache and physical memory contents. Therefore, cache contents are read/written only by accessing the memory locations shadowed by cache. However, since delays between introducing new data (writing) to a cached location and when that data is actually committed to physical memory can be taken advantage of to store data in cache while keeping it out of physical memory. A procedure for achieving this is:

- Configure memory location as type WB (Write-Back)
- Read memory location (cache lines are filled)
- Write critical information to memory location (only cache is written)

Following this, Reads from the memory location will return the critical contents from cache. When done, the cache can be overwritten and invalidated. Using this technique, the critical information is never written out to physical memory. This has the distinct advantage of not compromising the critical information to a bus-analyzer, as well as not providing a possibility for physical memory to be polled or extracted from the main board for analysis.

Keys or decrypted data may be manipulated in cache memory using the above technique. For keys, either data-cache or instruction-cache may be used: by storing keys directly in the former case or setting up an instruction sequence that generates a key in the latter.

However, there are several limitations worth mentioning. Storage of critical information in cache, in the interim where it does not get written through to memory, can only be maintained temporarily, since most cache invalidation procedures that occur internally will cause cached data to be written out to physical memory.

Furthermore, cached locations may be read by any process that has access to the address space being cached. Therefore, other processes that gain CPU control while the cache contains critical data may, in theory, obtain access to this data.

**Registers**

Registers are an appropriate storage location for keys, since they are located deep in the CPU domain and are never implicitly written out to physical memory. Not all registers are suitable for storage of decryption keys. Most contain values that have significant implication on execution flow, such as general purpose registers, registers that point to significant memory locations or registers that contain operational flags.

The Intel architecture includes registers under two major categories:

- Basic Program Execution Registers
- System-Level Registers

Truly Protect focuses on the latter, since most system-level registers are protected from user-applications and may only be accessed from Kernel-mode (privilege level 0). This provides better control over the possibilities for keeping keys locked in CPU and out of reach of malicious code. As mentioned above, system-registers that are suitable for storing arbitrary data without affecting execution flow are the best potential candidates for key storage.

Debug Registers

The Intel architecture contains 8 debug registers (DR0-DR7). DR6 and DR7 are used to report and configure breakpoint conditions; DR4-DR5 are reserved and DR0-DR3 are used to store required breakpoint addresses. Since it can safely be assumed that debugging breakpoints will not be used (and will actually be prohibited) while Truly-protect actively protects a system, the 4 breakpoint address registers, DR0-DR3, can be used to store a key. In a 32bit system, each of DR0-DR3 is 32 bits wide. Therefore, this totals 128 bits of key information.

To ensure that a breakpoint does not occur at some arbitrary address, which happens to be part of the truly-protect key, the DR7 register is configured to disable the 4 DR0-DR3 breakpoints. An extremely useful facility is the DR7.GD[bit 13]. If this bit is set a #DB exception is generated if any of the debug registers (including DR7) are accessed. While not enough to guarantee that no other Kernel-mode program maliciously gains access to DR0-DR3, this facility may be used to control such access as part of a larger key-protection scheme.

Model Specific Registers (MSRs)

The MSRs are a group of system-registers used to report or configure a variety of system-related attributes. They may be used, amongst others, to control debug extensions, performance-monitoring, machine-check and memory type range definition (MTRRs). The majority of these registers cannot be used to store arbitrary values, however we will seek those that can.

Different Intel processor families have slightly different MSRs, so that MSR usage needs to rely on their availability in the current system. This can be verified programmatically with the CPUID instruction.

Performance counters are the most readily available MSR registers for storage. The most basic Intel architecture contains two 32 bit counters. Truly protect takes advantage of the performance counters during the validation process. However once that is complete, the counter registers can freely be used to store decryption keys. Counter register load commands of arbitrary values are supported in Kernel-mode and their corresponding control-registers can be configured to disable counting, thus ensuring that the preloaded values do not change. Both counters total 64 bits of key information.

**Dynamic Keys**

Dynamic keys are keys, or key modifiers, that are computed temporarily at run-time. They are computed in close propinquity to where they are needed for decryption and then immediately disposed of. Therefore, in a sense they are not stored anywhere, beyond the short period of time when used for decryption. Consequently, they may be stored in any of the CPUs general-purpose registers, provided that no other task can gain access to the CPU during that time.

Dynamic-keys have the distinct advantage of not needing any prolonged storage, therefore no need to find a hiding place somewhere inside the CPU, unreachable to malicious programs. However, while that being true, the code required to compute the key does need to be stored somewhere and because code is relatively large (compared to a key) it must be stored in memory rather than in some internal register in the CPU. While this may seem like a tombstone for that idea – not all is lost. It may still be possible to write a dynamic-key calculation routine, whose instructions are not secret – rather its execution is controlled such that the calculation will be correct only if invoked by a legitimate source.

Two useful tools may be recruited for this task. The first is to make use of the performance counters to count HW side-effects in the process of generating a dynamic-key. The advantage of this is clear: Dynamic-key values cannot be calculated by reverse-engineering the calculation routine. The routine must actually be executed on the target machine in order

to achieve the correct results. The second is the validation process, which runs just before any keys are introduced into the system. The validation process guarantees a clean (of malicious code) system. This gives us an opportunity to setup a software "mouse trap" around our dynamic-key calculation routine. Taking this simile a step forward: the rational is that if the mouse (malicious program) goes for the cheese (calculation routine) the trap (exception) is triggered. Dissimilar to the real mouse-trap case, the software incarnation can either catch and be rid of the mouse or it can annihilate the cheese, and so to speak, leave the mouse hungry.

To sum, dynamic-key generators may be used to render decryption key material during run-time, alleviating the need for protected key storage – provided that the generator code cannot be reverse-engineered and its execution is controlled.

**Avoiding Kernel-mode Plug-ins**

A common problem associated with all the above proposed key storage locations is their potential vulnerability to malicious kernel mode drivers. The Truly-Protect system suggests that a validity check shall be carried out as part of the encryption and key setup procedure. The validation verifies that the target system is real (non-virtual machine), running a recognized O/S version and does not contain malicious Kernel-mode drivers. In other words, it is safe to install the encrypted version of the software in the target's memory and store the decryption keys in the CPU domain. From this point on the encrypted software executes while simultaneously being decrypted by the Truly-Protect JIT decrypt engine.

If at any point a malicious Kernel-mode driver is plugged in to the system while the protected software is executing, that driver may access the key storage locations, acquire the keys and use them to decrypt and obtain the protected software.

To successfully protect the keys, the Truly-Protect system must either completely prevent plugging in Kernel-mode drivers while the protected software is executing or obliterate the keys if such a plug-in occurs. The Windows O/S, for example, supports driver plug-in as a standard procedure, therefore it is assumed to be difficult to enforce complete driver-load prevention. Consequently, the authors believe that the latter alternative, calling for key obliteration in the event that a Kernel-mode driver is loaded while the Truly-Protect system is active – is a more realistic approach. This warrants that the protection system be aware of any attempt to add a new Kernel-mode driver to the system and be alerted in time to obliterate the key.

The success of this approach also heavily relies on the quality of the validation process, which must substantiate a "clean system", in the sense that no malicious Kernel-mode drivers exist at the time the key material is generated.

## 4. Conclusions and Future Work

The Truly-protect software-only protection system is based on executing encrypted software by decrypting it just-in-time during execution. Every execution unit (instruction or routine) is decrypted at the moment it is needed and the decrypted incarnation is purged immediately upon its completion. To achieve this, decryption keys must be present during runtime and the decryption keys must be hermetically guarded from malicious programs.
The internals of the CPU are considered the safest place to store and guard the keys. Therefore, Truly-protect is designed to use the keys for decryption without the keys ever leaving the internal confines of the CPU. This means they do not exist in memory and are never present on any of the external system buses.
Several storage places, inside the CPU, were considered:

- Cache – has an appropriate storage state which may contain values that are different from the memory it shadows. However, since this state is highly instable it can only be utilized for short periods.
- System Registers – are an appropriate storage place that is protected from all application level programs. However, is susceptible to prying by malicious kernel-mode drivers.

- Dynamic-Keys – these do not need prolonged storage (beyond the decryption process). Nevertheless, the code required to generate the dynamic-keys must be protected against malicious invoking.

Storage of key material inside the CPU domain is safe from User-mode programs but not from Kernel-mode drivers. Means must be provided to safe-guard keys from malicious drivers that may already exist in the system or are loaded while protected software is executing.

No single solution amply solves all aspects of protecting keys on a target system, such that they cannot be confiscated by malicious code. Our future and on-going efforts are focused on combining several such solutions in order to provide fully-protected, software only, DRM solutions.

## References


[1] A. Averbuch, M. Kiperberg, N. Zaidenberg. An Efficient VM-Based Software Protection. In NSS 2011.

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

# VII

## SYSTEM FOR EXECUTING ENCRYPTED JAVA PROGRAMS

by

Kiperberg, M.; Resh, A.; Algawi, A.; Zaidenberg, N.J. Submitted

*38th IEEE Symposium on Security and Privacy (IEEE S&P 2017)*

# VIII

## SYSTEM FOR EXECUTING ENCRYPTED JAVA PROGRAMS

by

Kiperberg, M.; Resh, A.; Algawi, A.; Zaidenberg, N.J. 2017

*3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*