Jyväskylän yliopiston julkaisuarkisto
Jyväskylä University Digital Archive

UNIVERSITY OF JYVÄSKYLÄ

This is an electronic reprint of the original article.
This reprint *may differ* from the original in pagination and typographic detail.

Author(s): Resh, Amit; Kiperberg, Michael; Leon, Roee; Zaidenberg, Nezer J.

Title: Preventing Execution of Unauthorized Native-Code Software

Year: 2017

Version:

Please cite the original version:

Resh, A., Kiperberg, M., Leon, R., & Zaidenberg, N. J. (2017). Preventing Execution of Unauthorized Native-Code Software. International Journal of Digital Content Technology and its Applications, 11(3), 72-90. http://www.globalcis.org/jdcta/ppl/JDCTA3804PPL.pdf

All material supplied via JYX is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

Preventing Execution of Unauthorized Native-Code Software

[1]Amit Resh, [2]Michael Kiperberg, [3]Roee Leon, [4]Nezer J. Zaidenberg
[1] *Deparment of Mathematical IT, University of Jyväskylä, Finland, amitr44@gmail.com*
[2] *Faculty of Sciences, Holon Institute of Technology, Israel, mkiperberg@gmail.com*
[3] *Deparment of Mathematical IT, University of Jyväskylä, Finland, roee.leonn@gmail.com*
[4] *School of Computer Sciences, College of Management, Israel, nzaidenberg@me.com*

Abstract

The business world is exhibiting a growing dependency on computer systems, their operations and the databases they contain. Unfortunately, it also suffers from an ever growing recurrence of malicious software attacks. Malicious attack vectors are diverse and the computer-security industry is producing an abundance of behavioral-pattern detections to combat the phenomenon. This paper proposes an alternative approach, based on the implementation of an attested, and thus trusted, thin-hypervisor. Secondary level address translation tables, governed and fully controlled by the hypervisor, are configured in order to assure that only pre-whitelisted instructions can be executed in the system. This methodology provides resistance to most APT attack vectors, including those based on zero-day vulnerabilities that may slip under behavioral-pattern radars.

Keywords: *Hypervisor, Trusted computing, Whitelisting, Attestation, APT-protection, Cyber-security*

1. Introduction

An abundance of malicious software attacks plague the computer software industry. The attack methodologies are diverse, ranging from code-injection, buffer-overflow, viruses, worms and Trojans to rootkits. Malicious code is usually designed to gain access to and steal the victim's data, such as personal information, credentials, trade secrets, or to gain access to the victim's system in order to take advantage of the resource for inflicting further damage. Malicious code motivation is predominantly financial but in some case other motivations may exist as well.

In many cases malicious attacks are not carried out in a single shot. Many attacks are multi-faceted, containing several intermediate steps, each designed to progress the offender to the next level of penetration before reaching the final goal. As an example, [1] details 5 stages of a Web malware attack leading from entry to execution on the compromised system:

- Entry – malicious code enters the victim system as a result of a drive-by download occurring when visiting a hacked site or following a malicious link in an email.
- Traffic Distribution – drive-by downloads execute inside browsers. Their primary goal is to download an exploit kit. Traffic redirection occurs to conceal the IP address from which the exploit kits are eventually downloaded.
- Exploits – once an exploit kit is downloaded it attempts to locate a system vulnerability that it can exploit in order to progress the attack. Exploits are usually encapsulated in PDF, FLASH, Java, JS or HTML files.
- Infection – once a vulnerability is found by the exploit kit, it is used to download the actual malware's executable code. SophosLabs identify several common malware payloads: Zbot(Zues) – steals personal information by logging keystrokes and grabbing display frames; Ransomeware – restricting access to the user's resources and demanding payment to restore access; PWS – steals user credentials and allows remote access; Sinowal(Torpig) – installs a rootkit to steal credentials and allow remote access; FakeAV – a Fake antivirus that "finds" fake viruses and demands payment to "clean" them out.
- Execution – the downloaded malware has been installed in the victim system and is executed. This is the stage where the actual damage is inflicted.

Other types of attacks exist as well, each seeking to abuse system or human vulnerabilities in order to inflict damages, gain access to privileged information or completely take control. Many of these attacks are similarly multi-stage. Attacks may exploit all or some of the following common stages:

- Entry – malicious code enters the system as a result of a malicious email attachment, a bogus executable installation a buffer-overflow, a USB disk insertion, a worm or a virus spreading.
- Non-privileged execution – in this mode of execution, malicious code that has entered the system executes in a low privileged level. It may still inflict some damage, however that damage is usually limited and may eliminate its capability to achieve persistency. In that case, the malicious code will disappear when the system is rebooted.
- Escalation: privileged execution – a much more hazardous case occurs when an unprivileged code exploits a system vulnerability (usually in the OS) and manages to escalate its privilege. It is beyond the scope of this text to describe the mechanisms that may be employed to achieve this, but the statistics are most staggering. Malicious code that gains privileged access may freely write to the filesystem on disk, to the main memory – both to user and to OS space, to the system registry or even to the boot record or BIOS memory.
- Acquiring Persistency – using the capabilities of privileged execution, malicious code can strive for persistency. In other words, the capability to survive system reboot as well as a complete system power-cycle. Achieving this level is the first step in "securing" the malicious code's survival in the compromised system. Many infections will also go to great lengths to camouflage their existence using a variety of methods, some very cunning, to avoid detection and removal.
- Compromised system – once malicious code has persistent execution on the system the perpetrator can potentially steal sensitive data, log keyboard activity to steal messages or passwords, grab screenshots or even achieve full remote-control of the system.

While system penetration is possible to some extent, without resorting to execution of unrecognized instructions in the system – ultimately all penetration goals are served only by executing some form of (rogue) executable instructions, which were not part of the system before the penetration. The methodology proposed by the authors in this paper, takes advantage of this fact, to provide an efficient way to protect against most such invasions, performed by a large variety of penetration techniques and also in many cases that utilize a previously unknown zero-day vulnerability.

The authors propose an approach whereby native-code is verified just before it receives execution rights. To achieve this, the entire system is first "whitelisted" by generating a database that contains signatures for every executable code-page that exists in the system's executable files, DLLs, drivers etc. A hypervisor is utilized to intercept and verify every execution attempt, at a page granularity, according to the whitelist database. The system is based on the approach proposed by Averbuch et al. [2] [3], in which an attested kernel module is responsible for performing cryptographic operations.

Hypervisors have been previously used to secure systems. For example, the Software-Privacy Preserving Platform (SP^3) [4] utilizes a hypervisor to maintain isolated memory-pages in ***protection-domains***. Physical pages in the system can be individually encrypted with a symmetric-key, where each domain has an associated set of keys whose pages it is allowed to use. The hypervisor intercepts interrupts and exceptions and uses shadow page-tables to manage decryption and encryption of the appropriate pages when the application shifts between domains. This methodology keeps domain access to protected pages isolated from other domains as well as from the OS. The hypervisor stores the key-database and domain key-associations in its own isolated memory. We have previously extended Truly-Protect hypervisor to support digital rights protection for digital video distribution. [5] [6] This is our second extension of TrulyProtect Hypervisor.

2. Thin hypervisor

A hypervisor, also referred to as a Virtual Machine Monitor (VMM), is software, which may be hardware assisted, to manage multiple virtual machines on a single system. The hypervisor virtualizes the hardware environment in a way that allows several virtual machines, running under its supervision, to operate in parallel over the same physical hardware platform, without obstructing or impeding each other. Each virtual machine has the illusion that it is running, unaccompanied, on the entire hardware platform. The hypervisor is referred to as the Host, while the virtual machines are referred to as Guests.

Hypervisors have been in use as early as the `60s on IBM mainframe computers [7]. After 2005 Intel and AMD introduced hardware support for virtualization (Intel VT-X [8], AMD AMD-V [9]) which allows implementing hypervisors on the ubiquitous PC platforms.

In order to support multiple OS guests, a hypervisor must unobtrusively intercept OS access to hardware resources so it can attend to them itself. The hypervisor can then manage hardware allocations that maintain proper separation between the Guests. The Guest OS is unaware of the hypervisor's intervention, as it experiences a normal hardware access cycle. The only distinction being the elapsed time, since the hypervisor mediation has a time-toll.

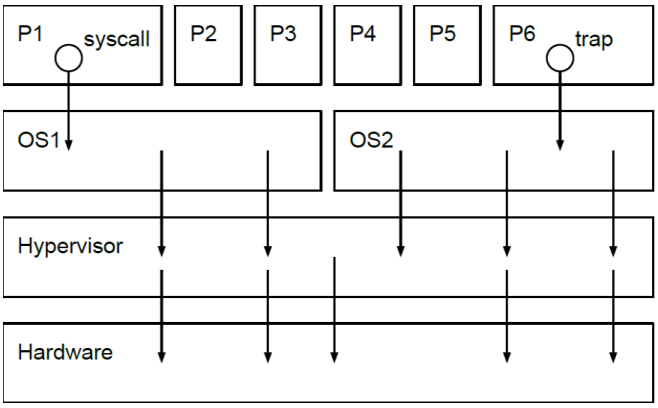

Figure 1. Virtualized system featuring a hypervisor and two operating systems executing 6 programs. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions and other interrupts, transfer control from user mode applications to their operating system. The operating system handles these conditions by requesting some service from the underlying hardware. The hypervisor intercepts those requests and handles them according to some policy.

To intercept OS hardware access, hypervisors can be configured to intercept privileged instructions, memory access, interrupts, exceptions and I/O, which are the OS vehicles for hardware access. Executing an intercepted privileged instruction causes a hypervisor VM_EXIT. In other words, the Guest is exited and the configured hypervisor intercept-routine is executed. When this occurs, the CPU mode changes from Guest-mode to Host-mode.

Guest applications that require hardware resources, execute system calls to request support from their OS. Figure 1 depicts this chain-of-execution for a hypervisor with two Guest stacks. After fulfilling the intercept, the hypervisor indiscernibly returns to the Guest. While hypervisors were generally designed to serve as virtual machine monitors, hypervisors, which control the underlying hardware platform, are also very good platforms to serve as software security facilitators.

The authors propose to use a hypervisor environment for securing a single Guest stack. Rather than wholly virtualizing the hardware platform, a special breed of hypervisor, called a ***thin-hypervisor***, is used [10] [11]. The thin-hypervisor is configured to intercept only a small portion of the system's privileged events. All other privileged instructions are executed without interception, directly, by the OS. The thin-hypervisor only intercepts the set of privileged instructions that allows it to protect an internal secret (such as cryptographic key material) and protect itself from subversion. Figure 2 depicts a thin-hypervisor supporting a single Guest stack. Since the thin-hypervisor does not control most of the OS interaction with the hardware, multiple OSs are not supported. However, system performance is kept at an optimum.

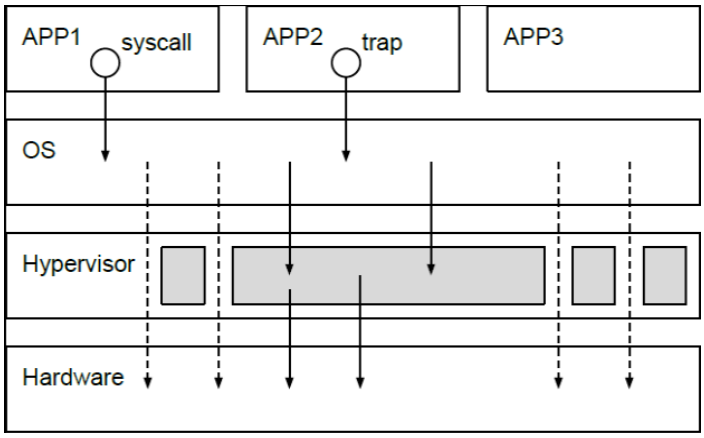

Figure 2. Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service from the underlying hardware. A thin hypervisor can intercept some of those requests and handle them according to some policy.

Thin hypervisors have been previously used for security purposes. TrustVisor [12] is a thin hypervisor that enables isolated execution of designated portions of an application. TrustVisor is booted securely by making use of a TPM chip and once in operation, it depends on hardware virtualization to isolate portions of memory with Secondary Level Address Translation (SLAT) as well as protect memory from DMA access by physical devices with DEV or IOMMU. TrustVisor utilizes this capability to (i) protect itself; and (ii) extend TPM facilities to a so-called µTPM environment that is used to provide high-speed trusted-computing primitives. These capabilities are further used by TrustVisor to achieve its ultimate goal of supporting a totally-isolated execution environment for designated self-contained software routines, called PALs (Pieces of Application Code). Software developers designate the portions of their codes that require isolation and group them into appropriate PALs. The developers register the PALs by providing a description of PAL bounds as well as memory regions they need to access. The TrustVisor guarantees that when PALs are called they operate in an isolated memory environment until they are exited.

A thin-hypervisor facilitates a secure environment by:

1. Setting aside portions of memory that can be accessed only when the CPU is in Host mode
2. Storing cryptographic key material in privileged registers and
3. Intercepting privileged instructions that may compromise its protected memory or key material

A thin-hypervisor is less susceptible to being hacked as a result of vulnerabilities, since its code and complexity are greatly reduced, as compared to a full-blown hypervisor.

Once this environment is correctly setup and configured, the thin-hypervisor can be utilized to carry out specific operations, which may include use of the internally stored key material, in a protected region of memory. As a result of the tightly configured intercepts and absolute host control of select memory regions, this activity can be guaranteed to protect both the secret key material and the operations' results.

The thin-hypervisor can effectively protect the secret key-material, after it is safely stored in privileged registers and the thin-hypervisor is correctly configured and active. However, the procedure by-which the secret material gets stored while the thin-hypervisor is being setup – is delicate business, since an adversary can potentially grab the secret at that point. An additional question, requiring an answer, is where the secret is kept while the thin-hypervisor is not active?

The authors' approach to solving these issues is based on an approach described in [13] and is comprised of the following principles:

- While the thin-hypervisor is not active, the secret key material shall not be stored anywhere in the system
- When setting up a thin-hypervisor, an external system shall be used to verify that the thin-hypervisor has control over the underlying hardware
- The same external system that verifies the thin-hypervisor shall provide the secret key-material

The first principle is important to rule out the possibility of keeping secret material under the cover of obfuscation, which is known to be ultimately vulnerable. The second and third principles require maintaining a remote key-server system and equipping it with the facilities to verify that a thin-hypervisor on a remote system has been properly setup and configured, such that a trusted environment is primed and can accept secret material.

2.1 Adversary Model

We assume that an adversary is freely able to access system memory for writing and reading. Memory can be accessed for writing in a variety of ways. For example, contents can be loaded from disk, arrive over a communication channel or be injected directly into memory by an executing application. We further assume that an adversary is also able to write to some memory regions that should in principle be protected by the OS, based on exploiting system vulnerabilities. Such regions include, but are not limited to, application code, privileged kernel-mode code and system drivers. Accordingly, memory that has been accessed for writing, by the application or by the OS, is never trusted for execution purposes.

Furthermore, it is assumed that an adversary cannot obstruct the operation of a root (primary) hypervisor that is based on hardware virtualization, as well as secondary memory translation (i.e., EPT) and IOMMU that operate at a privilege that is higher than the OS when a hypervisor is active.

Adversary attacks that are based on manipulating pure data in memory, in such a way as to render legitimate code malicious (referred to as code-reuse) are not considered.

2.2 Contribution

The authors propose a methodology and system that achieve a strong system-wide protection against execution of a wide array of unauthorized code penetrations. Our approach is distinguished from previous efforts by the implementation of an attested thin-hypervisor, which launches in an existing OS and which extends its security model over existing legacy applications without requiring their modification.

The unique approach described here allows a system to dynamically shift between protected and unprotected modes of operation. This situation can be appreciated, for example, in a BYOD situation, where enterprise employees can use their own computers for private (unsecure use) without enduring the performance overhead associated with protection, then shifting dynamically into protected mode to run office applications that require tight security. Applications that execute in protected mode, shall be protected and isolated from malicious code the computer may have contracted.

Dynamically shifting into protected mode is based on the capability to activate a thin-hypervisor after an OS already prevails. Securing trust in this situation entails administering a remote attestation procedure to establish a trusted environment in an otherwise untrusted computer system.

3. Achieving trust in a remote system

The problem of remote software authentication, determining whether a remote computer system is running the correct version of a software, is well known [14] [15] [16] [17] [18] [19]. Equipped with a remote authentication method, a service provider can prevent an unauthenticated remote software from obtaining some secret information or some privileged service. For example, only authenticated gaming consoles can be allowed to connect to the gaming networks [20] [21] [22] and only authenticated bank terminals can be allowed to fetch records from the bank database [23].

The research in this area can be divided into two major branches: hardware assisted authentication [24] [25] [26] and software-only authentication [14] [15] [27]. In this paper we concentrate on software-only authentication, although the system can be adapted to other authentication methods, as well. The authentication entails simultaneously authenticating some software component(s) or memory region, as

well as verifying that the remote machine is not running in virtual or emulation mode. Software-only authentication methods may also involve a challenge code that is sent by the authentication authority, and executed on the remote system. The challenge code computes a result that is then transmitted back to the authority. The authority deems the entity to be authenticated if the result is correct and was received within a predefined time-frame. The underlying assumption, which is shared by all such authentication methods, is that only an authentic system can compute the correct result within the predefined time-frame. The methods differ in the means by which (and if) they satisfy this underlying assumption.

Kennell and Jamieson proposed [14] a method that produces the result by computing a cryptographic hash of a specified memory region. Any computation on a complex instruction set architecture (Pentium in this case) produces side effects. These side effects are incorporated into the result after each iteration of the hashing function. Therefore, an adversary, trying to compute the correct result on a non-authentic system, would be forced to build a complete emulator for the instruction set architecture to compute the correct side effects of every instruction. Since such an emulator performs tens and hundreds of native instructions for every simulated instruction, Kennell and Jamieson conclude that it will not be able to compute the correct result within the predefined time-frame. The method of Kennel and Jamieson was further adapted, by the authors, to modern processors [13]. The adaptation solves the security issues that arise from the availability of virtualization extensions and multiplicity of execution units.

Establishing a thin-hypervisor that receives a remote secret (cryptographic key) in confidence and which may execute cryptographic operations with that secret key, provides an excellent software-only platform to utilize and sustain trust. The utilization of trust is based on being able to deliver encrypted or cryptographically-signed material to the remote system. The thin-hypervisor can decrypt and/or validate the received material and act accordingly. Any attempts to make changes, additions or deletions to the delivered material will inevitably be detected by the thin-hypervisor, provided the secret key is kept secret. Trust sustainability is upheld by eliminating any possible access to the secret material as well as rejecting any attempts to disrupt the code or state of the thin-hypervisor. Fortunately, a hypervisor has the available facilities to achieve just that.

Setting up a trusted thin-hypervisor on a remote system, while adhering to the 3 principles noted in the previous section, involves the following validations:

1. The thin-hypervisor's code is validated
2. The validated code is the one that executes when a VM_EXIT occurs
3. The thin-hypervisor controls the underlying hardware

3.1 Overview of the methodology

The vehicle to perform this remote verification is a piece of code, called an attestation-challenge [28] [29]. The attestation-challenge is administered by the key-server to the remote machine, as it is configuring the thin-hypervisor. The remote machine is required to load and execute the challenge code, returning an attestation result to the key-server within a pre-limited time-frame. The attestation-challenge calculates the checksum of the thin-hypervisor code, but in addition convolutes the checksum calculation with hardware side-effects, sampled by the challenge as it is executing. The side-effect samples are hardware-registers that count hardware events, such as cache hits or misses, TLB hits or misses etc.

The key-server considers a correct response received within the allotted time-frame, proof that the correct thin-hypervisor code is executing and it has true control of the remote system's hardware. Upon receiving a correct response the attestation can provide keys that the attested computer hides and protects in its CPU. [30]

3.2 Remote attestation

As described above, the attestation challenge calculates the checksum of the thin-hypervisor's code convoluted by hardware event samples. The attestation challenge is composed of several computational nodes. Each node executes a single operation related to the challenge result calculation and then branches to the next node according to the current result value. Three different branches are possible for each node:

- Branch A: if the result parity is even (50% chance)
- Branch B: if the sign bit is set (25% chance)

- Branch C: Otherwise (25% chance)

Branch target nodes may be the same or different, for each possible branch option. The variety of nodes include:

- Checksum operation – Sum a hypervisor code value
- XOR hardware counter – Xor hardware-event-counter *i* with current checksum result
- AND hardware counter – AND hardware-event-counter *i* with current checksum result
- Multiply hardware counter – Multiply hardware-event-counter *i* with current checksum result
- MAC calculation (such as SHA-1)

  where *i* is a Data-Cache Hit, Data-Cache Miss, TLB-Hit, TLB Miss, etc. Due to the multiple branches stemming from each node, the entire set of nodes comprises a network.

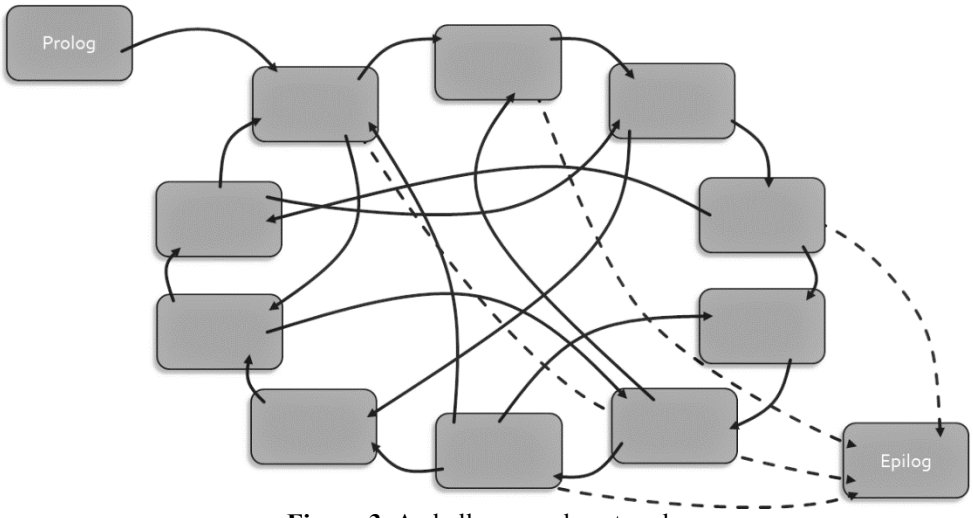

Figure 3. A challenge node network.

The node network is built to guarantee that every circuit contains at least one of each node-type. The first node to execute is the "Prolog" node, which sets up the environment and configures the hardware side-effect counters. The "Epilog" node is the last node to execute. It performs clean-up and returns the final challenge result.

Checksum calculation is performed by summing a wide virtual space that is redundantly mapped to the physical memory space that contains the code regions need to be attested along with their page-tables. The challenge is always accompanied by a (pseudo-random) *virtual map* that is designed to map the relatively small physical-page region to the relatively large virtual space. Naturally, each physical-page is mapped to multiple virtual-pages. The physical-page region includes:

- The thin-hypervisor code pages
- The challenge code page (all the code of the nodes)
- The page-table pages that define the virtual map

The challenge nodes are contained in a single physical-page, however, individual nodes are mapped at different virtual space locations and as such, each Node executes from a different location.

The checksum calculation order is governed by a pseudo-random-walk according to an LFSR (Linear-Feedback-Shift-Register) generator [31]. Every virtual-space address is visited once, however, physical addresses are visited multiple times. This is designed to induce side-effects. In a check-summing node, the value at each address is accumulated to the checksum. Other node types perform additional action on the current result, such as adding in hardware event counter values or calculating a MAC.

The virtual-space random walk creates pseudo-random data-cache patterns that affect future cache hit/miss events. Similarly, execution of nodes, each at a different virtual location, creates pseudo-random code-cache and TLB cache patterns. Each affecting its corresponding cache hit/miss events. Hardware side-effect convoluting type nodes, incorporate a transient hardware counter result into the accumulated checksum. Thereby, both changing the current result value, as well as node progress flow.

It is stipulated that challenge results calculated in an environment that is different than the intended (for example at attempt to execute our thin-hypervisor under an emulator or as a nested-hypervisor) will

generate a significantly different challenge result and thus be easily detected. The possibility of calculating a correct result by means of emulation shall also be impossible within the allotted timeframe restriction. We previously shown the challenge server can generate and manage challenges effectively [32]

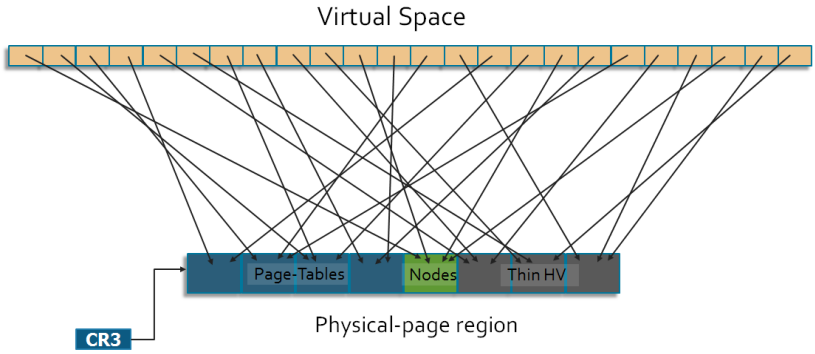

Figure 4. A challenge node network.

4. Controlled execution

4.1 Introduction

The x86 architecture allows the operating system to control memory access rights of applications through the virtual paging mechanism. Similarly, virtualization extensions, which were introduced by Intel and AMD, allow a hypervisor to control memory access rights of operating systems through a mechanism called Second Level Address Translation (SLAT). Intel and AMD refer to this mechanism as Extended Page Table (EPT) [8] and Rapid Virtualization Indexing (RVI) [9], respectively. Virtual paging and SLAT can be used to specify the "read", "write" and "execute" rights of a particular memory page ("execute" rights are controlled by the "NX bit" in virtual paging [8] Unlike virtual paging, SLAT defines the memory access rights of the physical rather than the virtual pages, thus providing the hypervisor with complete control over the access rights in all memory modes.

Our hypervisor uses SLAT to prevent execution of unauthorized software. Initially, the hypervisor forfeits the "execution" rights of all pages, thus effectively intercepting any execution attempt. Upon such intercept, the hypervisor verifies the executing page authenticity, by hashing the page content and comparing it to a precomputed value. After authenticity is established, the hypervisor grants the page "execution" rights but forfeits its "write" rights, thus intercepting attempts to modify authenticated pages. Upon interception of such a modification attempt, the hypervisor grants the page "write" rights but forfeits its "execution" rights. Therefore, at all times, a page can have either "execution" rights or "write" rights, but not both.

Page authentication in its simplest form consists of two steps: hashing and comparison. In the first step, the hypervisor applies a hash function to the page being authenticated. In the second step, the hypervisor checks whether the result of the hash function appears in a database of valid hash values. This database is built ahead of time by scanning the hard drive for installed applications, computing the hash values of the applications' code pages, storing the hash values in a database, and finally signing the database, in order to prevent its unauthorized modification. Section 4.2 contains a detailed description of the database structure.

In some cases, after loading a page into memory, the operating system alters the page's content according to a set of rules called relocations. A relocation describes an absolute address that is referenced by the application that might need to be adjusted. This adjustment is necessary only if the application was loaded to a non-preferred location, but this is usually the case [33] [34]. In order to apply a relocation at offset x, the operating system first computes the relocation offset, which is the difference between the application's actual and preferred loading locations, and then adds this difference to the address at offset x. Conceptually, during a page's authentication, the hypervisor first restores the original values at the relocation offsets, and then computes the hash of the resulting page. In practice, the page is not modified

during authentication; instead, the hashing calculation is performed on some temporal value at relocation offsets.

Unfortunately, some pages contain both code and data. Obviously, the hypervisor cannot fully authenticate such pages. On the one hand, granting these pages with "execution" rights will allow execution of any code in the unverified (data) area of the page, and therefore compromise the security of the entire system. On the other hand, the authentic code cannot be executed from a page without "execution" rights. We propose the following solution to this problem. The hypervisor grants the page with "execution" rights but starts monitoring the guest's instruction pointer. Whenever the instruction pointer exits the authenticated area, the hypervisor forfeits the "execution" rights of the page. Section 4.4 contains a detailed description of this process.

The hypervisor monitors the instruction pointer using the processor's debugging facilities. Specifically, the hypervisor resumes the guest in a single-step execution mode. In this mode, the processor generates an interrupt after every executed instruction, thus enabling the hypervisor to verify that only the authenticated portions of the page are executed, and thus maintain appropriate rights for partially authenticated pages. Some processors provide an extension to the single-step mode, in which the interrupt is generated only after execution of branch instructions, such as jumps, calls and returns. The instruction pointer can exit the authenticated area not only due to a branch instruction but also by falling through the last instruction. The hypervisor intercepts the latter case by installing a hardware breakpoint at the byte following the last instruction of the authenticated area.

4.2 Database structure

We begin our detailed explanation of the execution prevention mechanism, by describing the structure of the database that contains the hash values (see Figure 5). That database consists of modules descriptors. Each module descriptor contains information of a specific executable (PE file in Windows [35] or ELF file in Linux [36]) which resides on the machine. Each descriptor is signed by an RSA signature in order to prevent an attacker from manipulating its contents. We note that an attacker can potentially remove module descriptors, but he cannot alter existing descriptors or add new ones. Each module descriptor contains its size, which allows to move to the next descriptor. The descriptor also holds the path of the executable which is represented by this descriptor. The driver uses the path field to identify the descriptor corresponding to the loaded image. As was explained in section 4.1 the verification procedure needs to know the executable's expected location in memory. This information is stored in the "Base" field of the module descriptor.

Finally, the module descriptor contains a list of section descriptors. Each section descriptor corresponds to an executable section of the executable, and contains the following fields:

- Record size – the size of this section descriptor. This field allows to move to the next descriptor.
- Offset – the offset of the section described by this descriptor from the beginning of the image file.
- Length – the size of the section described by this descriptor.
- Page[i] – page descriptor that corresponds to the ith page of the section.
- # Relocs – the amount of relocation descriptors that follow.
- Reloc[i] – relocation descriptor – explained below.
- # Datums – the amount of the datum descriptors that follow.
- Datum[i] – datum descriptor – explained below.

The amount of page descriptors can be deduced as follows. Let L denote the section's offset rounded down to a page boundary and let R denote the sum of section's offset and section's length rounded up to a page boundary. Then the amount of page descriptors if $(R-L)/4096$. In other words, that database holds a page descriptor even for partial pages, i.e. pages that only partially belong to the section. In that case only the bytes that belong to the section are hashed.

The page descriptor consists of the hash value of the corresponding page (or its part), and two indexes to the Reloc[] array: the index of the first relocation and the index of the last relocation that apply to this page. The relocation descriptor consists of two fields: type – which determines whether the relocation applies to an 8-byte or a 4-byte region, and offset – the location in page where the relocation applies. The datum descriptor consists of two fields: offset – offset from the module beginning, value – 8 bytes at that location. The verification procedure uses the datum descriptor array (in addition to the relocation array) during verification of pages that contain relocations that cross page boundaries.

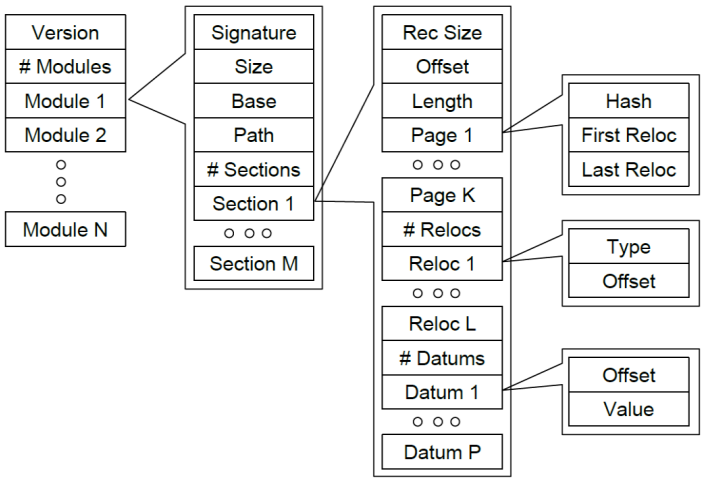

Figure 5. Structure of the database containing the hash values. The database consists of many modules, each of which consists of many sections. Each section contains the hash values of pages that it occupies, the relocations in those pages and datums – values of relocation that cross page boundaries.

4.3 Execution prevention

The hypervisor is part of a device driver, which acts as a mediator between the hypervisor and the operating system. In particular, the driver constructs some data structures that are later used by the hypervisor. We note that the hypervisor cannot (and does not) trust these data structures and therefore their critical parts contain a signature proving their authenticity. During initialization, the driver loads the database containing the hash values to a pageable region of memory, and installs two callbacks; the first callback is invoked when the operating system loads an executable to memory, the second callback is invoked when a process terminates. Both callbacks update a data structure that represents the memory layout of all the processes that are currently active. The data structure is a list of process descriptors. Each process descriptor contains the corresponding process identifier and a pointer to a list of module descriptors. Each module descriptor contains the location in memory of the corresponding module and the database index of this module's descriptor. Figure 6 depicts this data structure.

During the driver's initialization it installs the hypervisor which manages the access rights of physical pages. The hypervisor and the driver callbacks operate concurrently: the callbacks update the memory layout data structure that is used by the hypervisor. Unfortunately, the driver initialization order is determined by the operating system and cannot be affected. Therefore, the operating system may load and initialize some drivers prior to our driver initialization. Consequently, the callback, which is installed during initialization, will not be called on those drivers. Our driver solves the problem, by traversing operating system-specific data structures that contain information about the drivers that were loaded. Figure 7 presents the data structures that are used by a 64-bit version of Windows 8.

Initially the hypervisor forfeits the "execution" rights of all the physical pages. An attempt to execute an instruction triggers an "EPT Violation" (unauthorized access to physical memory) which passes the control to the hypervisor. The hypervisor verifies the authenticity of the page containing the instruction and changes its access rights to "read" and "execute". An attempt to write to this page triggers an "EPT violation" and the hypervisor changes the access rights to "read" and "write". This process is depicted in Figure 8. A detailed description of the verification procedure appears below.

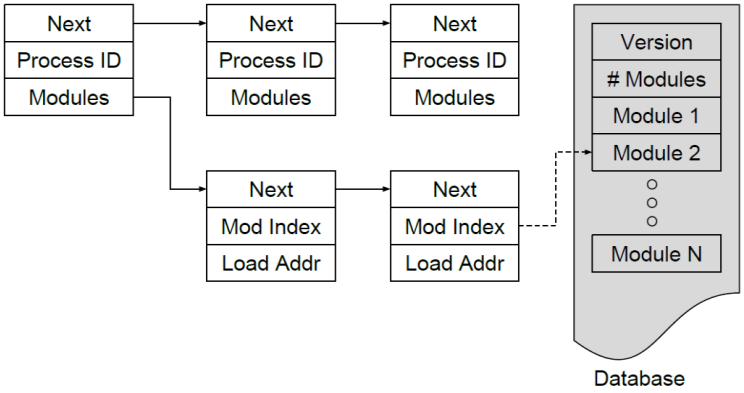

Figure 6. Memory layout data structure. The memory layout consists of a list of process descriptors. Each process descriptor contains the process identifier of the corresponding process and a pointer to a list of module descriptors. Each element of the module descriptors list contains the index of the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

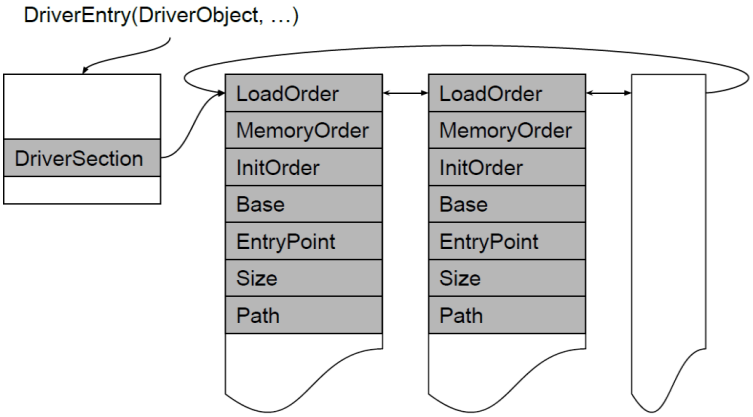

Figure 7. Memory layout data structure. The memory layout consists of a list of process descriptors. Each process descriptor contains the process identifier of the corresponding process and a pointer to a list of module descriptors. Each element of the module descriptors list contains the index of the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

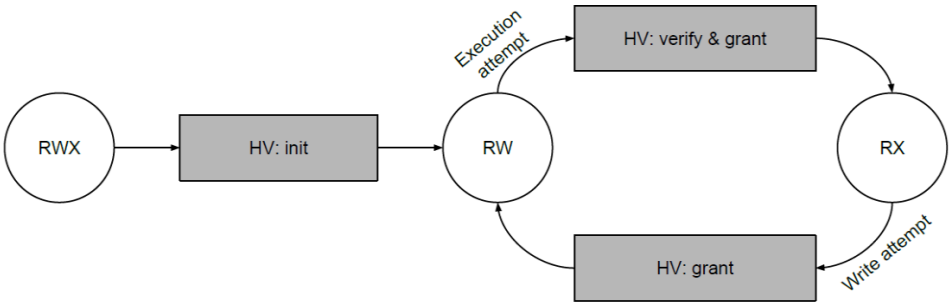

Figure 8. Physical pages access rights state diagram. "RWX" represents full access rights. "RW" represents "read" and "write" access rights. "RX" represents "read" and "execute" access rights.

On a multiprocessor system the hypervisor has a different configuration structure for each processor. In particular, each processor has its own EPT hierarchy, which can independently (of other processors) specify the access rights for each physical page. The hypervisor has to maintain identical configurations of all the EPT hierarchies (with a few exceptions, as we will see later) in order to prevent execution of unauthorized instructions.

Consider the following scenario: an authentic page request execution rights on processor A. The hypervisor verifies the page and grants it "read" and "execute" access rights, thus preventing its further modifications. However, processor B still has "read" and "write" access rights to this page, which enable it to modify the contents of this page. A malicious user can write malicious code to this page using processor B and then execute this malicious code on processor A.

Unfortunately, a processor can modify only its own EPT hierarchies [8]. Therefore, whenever the hypervisor on some processor decides to change the access rights of a page, it sends a request to hypervisors on other processors to make the intended change in their EPT. Only when all the EPT hierarchies of all the other processors were changed, the same change is made on the EPT hierarchy of the initial processor.

The request mechanism is implemented as follows. During its initialization the hypervisor allocates a constant-size queue of requests for each processor, which represents the access rights requests that the hypervisor running on that processor needs to serve. In addition the hypervisor installs an interrupt service routine on a special vector (0xFE), which is not in use by the operating system. The interrupt service routine issues a hypercall with a special value, which informs the hypervisor that its requests queue is not empty. The hypervisor serves this hypercall by applying all the changes described by the requests in the queue and clears the queue. In order to issue a request to another (remote) processor, the hypervisor performs two steps: (1) it inserts a new element to the requests queue of the remote processor, and (2) sends an IPI to the remote processor on the special vector (0xFE). After issuing the request, the hypervisor waits for the changes to be applied. Figure 9 depicts the entire process of access rights modification as it is performed on a multiprocessor system.

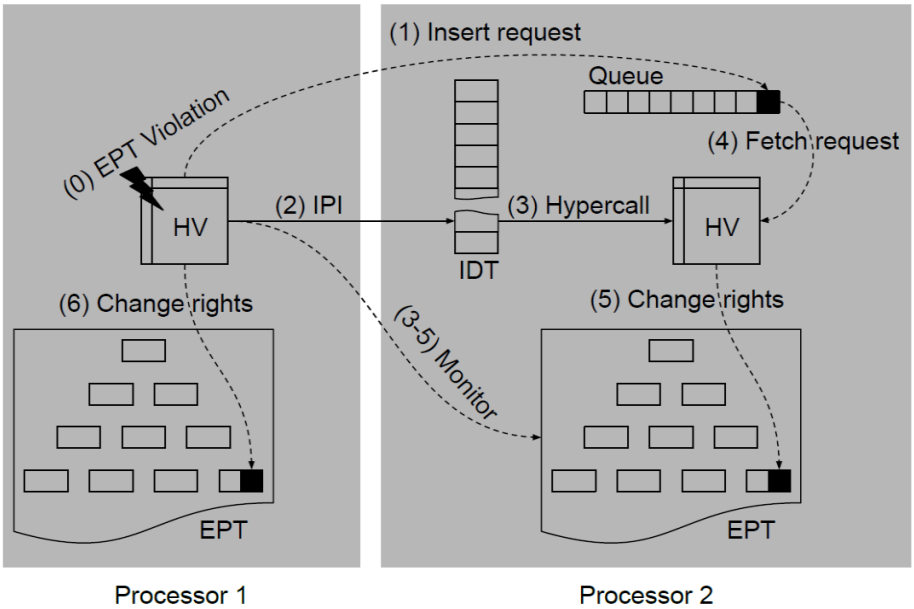

Figure 9. Access rights modification on a multiprocessor system: (0) an EPT violation on processor 1 triggers the hypervisor; (1) the hypervisor inserts a request into the request queue of processor 2; (2) the hypervisor sends an IPI to processor 2; (3-5) the hypervisor monitors the EPT hierarchy of processor 2 and waits for the change to occur; (3) the IPI that was sent in step 2 triggers an ISV; (4) the ISV hypercall to the processor 2 hypervisor; (5) the hypervisor fetches the request and changes the EPT hierarchy accordingly; (6) the processor 1 hypervisor observes that modification in the remote EPT hierarchy and performs the same modification in its local EPT hierarchy.

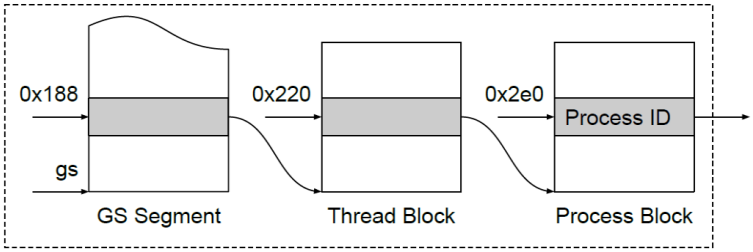

Figure 10. The GS register points to a local storage of the current processor. This local storage points to a data structures that represents the currently executing thread – the thread block. The thread block points to a data structure that represents the process which hosts the thread – the process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

The verification procedure can be seen as a boolean function returning true iff the verification succeeds. This function has one parameter – the virtual address that triggered the EPT violation handler. The function performs the following steps:

1. Fetch the current process identifier from OS-specific data structures. Figure 10 depicts this process on a 64-bit version of Windows 8.
2. Locate the process identifier in the memory layout data structure, which was prepared by the driver. The process descriptor contains a pointer to a list of module descriptors.
3. Locate the module descriptor that contains the virtual address that triggered the EPT violation handler. The module descriptor contains the index of the database entry that corresponds to this module.
4. Copy the module descriptor from the database to a memory region that is protected by an EPT (i.e. all types of access are restricted).
5. Validate the signature of the module descriptor.
6. Locate the information describing the page that triggered the EPT violation:
    a. Locate the section descriptor
    b. Locate the hash value of the page
    c. Locate the index of the first and the last relocations
    d. Locate the index of the first and the last datums
    e. Compute the address of the first and the last bytes described by the hash value. For example, if only the first 20 bytes of the page belong to the section, then only those bytes should be hashed.
7. Hash the page (or its part) as follows:
    a. Let p be a pointer to the first byte to be hashed
    b. Initialize pi to 0
    c. For each relocation r do:
        i. Hash the bytes [pi..r.offset-1]
        ii. Let d be the datum at offset r.offset
        iii. If d is null, fix the value at r.offset and hash it
        iv. Else, hash d.value and verify value at r.offset
        v. Advance pi to r.offset+r.length
    d. Hash the bytes [pi..the last byte to be hashed]
8. Compare the hash result to the expected hash value and return true iff they are equal

Figure 11 presents the most general example of a verification process. Datums hold the values of relocations that cross page boundaries. Since on the one hand the verification procedure must read the value at the relocation position but on the other hand it must not attempt to read data that may induce a page fault, we chose to store the values of relocation that cross page boundaries in a special array – the datums array.

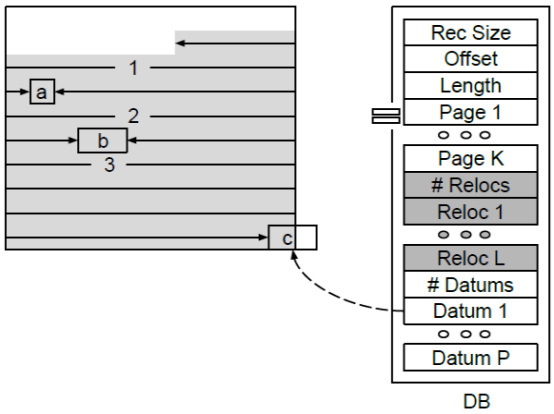

Figure 11. Page authentication in its most general form. In this case the section starts in the middle of a page. The section contains three relocations: *a*, *b* and *c*. Relocation *c* only partially belongs to the page being authenticated. The verification function first computes the hash of the bytes preceding relocation *a* (the first segment). It then subtracts from the value at position *a* the difference between the actual and the expected locations of the module and hashes the result. The same is done for relocation *b* and the second segment. Finally the verification function hashes the third segment and the relevant part relocation *c*. Since the value of relocation *c* cannot be read from the page, it is read from the datums array.

4.4 Secure execution of mixed pages

Some pages may contain both code and data. Usually, such pages appear on a boundary between a code section and a data section when those sections are not page-aligned. The problem with such pages is that on the one hand it is unsafe to grant these pages "execution" rights since they cannot be authenticated entirely, and on the other hand, the code in these pages cannot execute without "execution" rights. The solution to this problem is *controlled execution*. In essence after granting the page "execution" rights, we make sure that the control does not exit the authenticated area, by monitoring the instruction pointer. The hypervisor monitors the instruction pointer by activating the hardware debugger in a single-step mode. In this mode, the processor generates an interrupt on vector 1 after each instruction executes. The hypervisor intercepts this interrupt and checks whether the instruction pointer has left the authenticated area, and if so, the hypervisor forfeits the "execution" rights of the page.

The hardware debugger is controlled by the debug control register (DR7), the debug address registers (DR0-DR3) and the flags register. These registers define conditions in which the processor should generate a breakpoint, which is actually an interrupt on vector 1. When the defined conditions are met, the processor generates an interrupt and sets the debug status register to report the conditions that were sampled. A hypervisor can intercept interrupts and attempts to access the debug and the flags register. In other words, the hypervisor has full control of the debugging facilities and can, therefore, use these facilities securely, as will be described below.

In order to start monitoring the instruction pointer, the hypervisor sets the trap flag in the flags register and begins intercepting all interrupts (by modifying the guest IDT). After every instruction executed by the guest, a VM_EXIT occurs, enabling the hypervisor to check whether the instruction pointer is within the authenticated area. The processor clears the trap flag when an interrupt occurs, therefore the hypervisor must intercept not only the interrupt at vector 1 (the breakpoint vector) but also all the other interrupts. When an interrupt occurs, the hypervisor forfeits the "execution" rights of the partially authenticated page.

On modern processors we can improve the performance of the presented system. The IA32_DEBUGCTL MSR provides additional means to define the breakpoint conditions. Specifically when the *single-step on branches* flag (bit 1) is set (in addition to the trap flag in the flags register), the processor generates a breakpoint after every branch instruction, rather than every instruction. During instruction pointer monitoring, the hypervisor sets this flag thus intercepting all branches that may potentially transfer the control outside the authenticated area. Another way to leave the authenticated area is by falling through the last instruction. Therefore, the hypervisor installs a breakpoint on the byte

following the last instruction, by writing its address to DR0 and setting the appropriate flags in the debug control register.

4. Management station

The hypervisor that was described in section 2 can prevent execution of unauthorized software by exploiting the SLAT mechanism. Obviously, the hypervisor can do so only after its activation. Therefore, the system remains vulnerable before and during its initialization: a malicious software may acquire execution rights and then either activate a malicious hypervisor or prevent activation of our hypervisor. In both cases, our hypervisor cannot provide protection against execution of such an unauthorized software. It is, therefore, desirable to inform the user about the protection status of the given system.

The management station has two responsibilities: attestation and monitoring/notification. By attestation, we mean that the management station acts as the remote key-server, attests the hypervisor that is being activated on a remote system and provides it with some secret information (i.e., cryptographic key). A detailed description of this process appears in section 3. The attestation protocol guarantees that the secret information is provided only to authentic hypervisors, which can then protect the system from unauthorized access. Therefore, possession of this secret information is a proof of the possessor's authenticity.

The second responsibility of the management station is monitoring and notification, by which we mean that the management station constantly monitors and informs the user about the protection status of remote systems, for example by displaying the statuses on the screen. The hypervisor is obligated to send a periodic message to the management station, thus indicating that the remote system is protected. The hypervisor signs its messages with the secret information that it received from the management station during the attestation protocol.

In order to prevent replay attacks, the management station generates and sends to the hypervisor a random number s which acts as a session id. The session id s is sent only once during the attestation protocol. At the t's time unit the hypervisor sends to the management station a signed message containing (s,t). This message proves that the hypervisor belonging to session s is active at time t. Figure 12 depicts the described protocol.

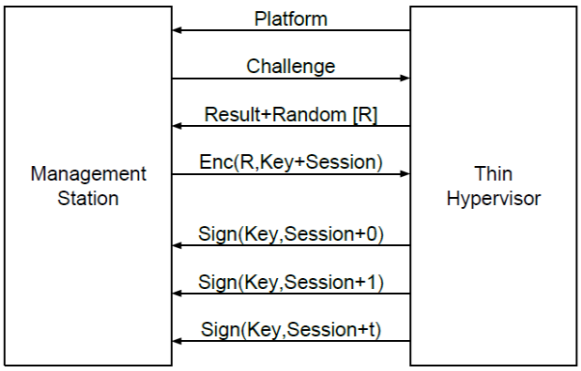

Figure 12. The protocol between the management station and the thin-hypervisor. The protocol consists of a 4-way handshake and periodic notifications. The "+" sign here means concatenation.

5. Performance

System overhead, as a result of execution protection, is attributed to actions that need to take place in the hypervisor during a VM_EXIT. This occurs when (a) execution of a write-only page is attempted and (b) as a result of a write to an execute-only page. The former's handling is more involved, since it warrants calculating the page's hash and verifying its signature, while in the latter case the operation is automatically granted. In both cases, however, the EPT needs to be updated. In single-processor environments, updating the EPT is straightforward, however, in multiprocessor environments, as

previously detailed, this is more elaborate, since it requires interrupting all the other processors by activating their respective hypervisor, which in turns updates its own EPT.

The (a) and (b) intercepts, mentioned above, occur when an executable page is first executed after the application was loaded and after a page was swapped out and then back in. Therefore, overhead is also closely related to the swap activity in the system.

Performance measurements of execution-protection overhead were conducted by measuring overhead directly as well as by running well-known benchmarks on single-processor and multiprocessor systems, with and without execution protection. The benchmark suite used was the "Phoronix Test Suite" [37]. A variety of test benchmarks were selected to reflect different types of loads, such as: CPU intensive, graphics, disk-access and network.

The tests were performed on a system with the following configuration:

- Intel Core-i7-3687U@3.3GHz (4 Cores)
- 8192MB DRAM
- Intel HD4000 Graphics
- Intel 82579LM Gigabit Network
- Linux (Ubuntu 14.04 kernel 3.19.0-25 generic X86 SMP)
- GCC 4.8.4

5.1 Test A

In the first test, we measure the direct overhead associated with authorizing a writable page for execution. An executable file is mapped to memory. The executable file contains a function `void f(void)` configured on a page boundary. The first instruction in `f()` is the return instruction; The Linux `posix_fadvise()` function is called to ensure that when `f()` is called a page fault requiring a page-load from disk shall occur. This also mandates a VM_EXIT and an executable-page validation when the system is execution-protected. We measure the number of CPU cycles involved in calling `f()`. We measure 10000 calls to `f()` while execution-protection is enabled and disabled. The average number of CPU cycles required to execute `f()` without execution protection enabled was: 917021, while with execution protection enabled was: 976754. The difference, 59733 cycles, reflects the number of CPU cycles required to authenticate a page for execution.

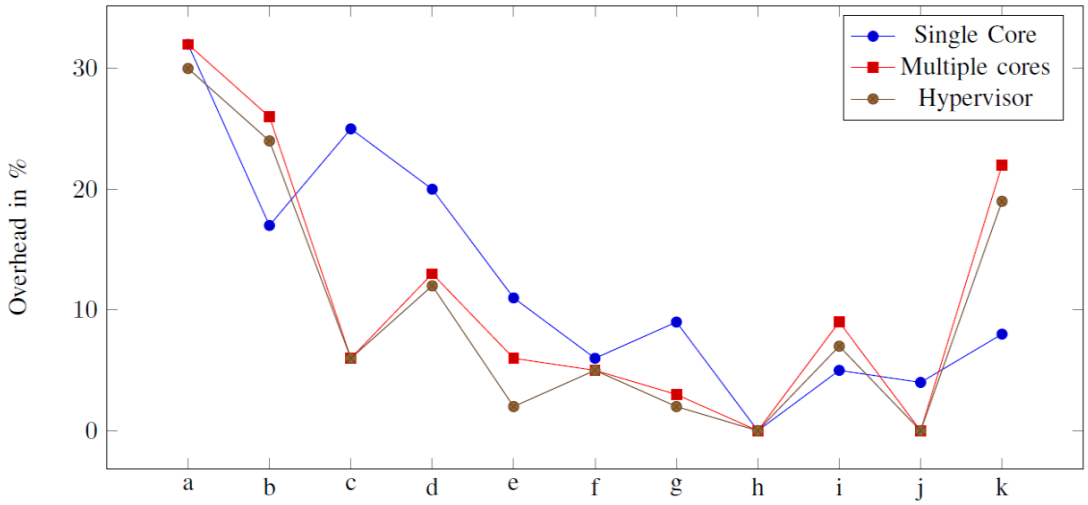

Figure 13. Overhead of the benchmark execution under different conditions: (a) single core; (b) multiple cores; and (c) with hypervisor but without execution protection

5.2 Test B

In the second test, we measure the overhead associated with executing intensive benchmarks selected from the "Phoronix Test Suite":

a) Apache – Static Web Page Serving
b) X11 – PutImage Square
c) X11 – Scrolling 500x00 px
d) X11 – Char in 80-char aa line
e) X11 – PutImage XY 500x500 Square
f) X11 – Fill 300x300 px AA Trapezoid
g) X11 – 500px Copy from Window to Window
h) X11 – Copy 500x500 Pixmap to Pixmap
i) X11 – 500Px Compositing from Pixmap to Window
j) X11 – 500px Compositing from Window to Window
k) Unpacking the Linux Kernel

To measure the effects of multiple cores, the benchmark comparisons were executed on a single core (by disabling other cores) and once again when all cores were enabled. In each case the benchmark was executed on a system with execution-protection enabled and disabled to generate the overhead comparison. The results are presented in Table 1 and depicted graphically in Figure 13.

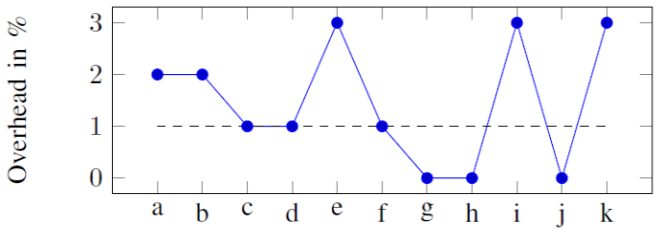

Figure 14. Overhead of execution protection only after subtraction of the hypervisor overhead. The dashed line represents the average overhead.

	Single	Multiple	Hypervisor	Net
a	32%	32%	30%	2%
b	17%	26%	24%	2%
c	25%	6%	6%	1%
d	20%	13%	12%	1%
e	11%	6%	2%	3%
f	6%	5%	5%	1%
g	9%	3%	2%	0%
h	0%	0%	0%	0%
i	5%	9%	7%	3%
j	4%	0%	0%	0%
k	8%	22%	19%	3%

Table 1. Test results

5.3 Evaluation

The results show that the total overhead of the execution-protection with a thin-hypervisor exists within a 0%-30% band, depending on the type of benchmark tested. When hypervisors are activated on systems and secondary level address translation (SLAT) is active, system overhead is caused by the additional translation required for memory access, which was measured as well. This parasitic overhead, as well as overhead caused by response to mandatory VM_EXIT events is associated with all hypervisors, however is minimized when using a thin-hypervisor. By subtracting this parasitic overhead from the general overhead values obtained for each benchmark, we present the net overhead associated with execution-protection, as can be seen in Figure 14 and in the rightmost column of Table 1. The results show an average overhead value of 1% within a 0%-3% range.

6. Conclusions

The growing threat of malicious code infiltration into computer systems is extremely grave in light of the economic losses and potential havoc they bestow. Hackers are becoming shrewder and much more cunning in their attack methodologies. They are winning the battle with the anti-malware protection industry, which is propagating an abundance of security software products geared to monitor, identify patterns and employ behavioral heuristics. As the authors point out, all Advanced-Persistency-Attacks (APTs) eventually need to execute instructions on the processor. Therefore, a suggested alternative method to eradicate most APTs is real-time monitoring and validation of executing instructions. An undertaking which can be appropriately addressed by using an attested, and therefore trusted, hypervisor. The associated total overhead is confined to 30%, where in most scenarios it is below 15%. With computer hardware performance advancing in great leaps, we believe that in return for rendering a system substantially safe from APTs, viruses, worms, buffer-overflows and malicious code injection, this overhead is justified.

References

[1] McCormack, "Five Stages of a Web Malware Attack," Sophos, Nov 2014. [Online]. Available: https://www.sophos.com/en-us/medialibrary/Gated%20Assets/white%20papers/sophos-five-stages-of-a-web-malware-attack.pdf.

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

[3] A. Averbuch, M. Kiperberg and N. J. Zaidenberg, "Truly-Protect: An Efficient VM-Based Software Protection," *Systems Journal, IEEE,* vol. 7, no. no. 3, p. 455–466, 2013.

[4] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Seattle, WA, USA, 2008.

[5] A. David and N. J. Zaidenberg, "Maintaining streaming video DRM," in *ICCSM*, Reading, 2014.

[6] N. J. Zaidenberg and A. David, "TrulyProtect video delivery," in *ECIW*, Jyvaskyla, 2013.

[7] R. J. Creasy, "The Origin of the VM/370 Time-sharing System," *IBM J. Res. Dev.,* vol. 25, no. no. 5, p. 483–490, 1981.

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

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

[10] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo and K. Kato, "Bitvisor: A thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, 2009.

[11] Y. Chubachi, T. Shinagawa and K. Kato, "Hypervisor-based Prevention of Persistent Rootkits," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, 2010.

[12] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

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

[14] R. Kennell and L. H. Jamieson, "Establishing the Genuinity of Remote Computer Systems," in *Proceedings of the 12th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003.

[15] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. v. Doorn and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2005.

[16] C. Castelluccia, A. Francillon, D. Perito and C. Soriente, "On the Difficulty of Software-based Attestation of Embedded Devices," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2009.

[17] D. Schellekens, B. Wyseur and B. Preneel, "Remote Attestation on Legacy Operating Systems with Trusted Platform Modules," vol. 74, no. no. 1-2, p. 13–22, Dec 2008.

[18] A. Seshadri, M. Luk, A. Perrig, L. v. Doorn and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *Proceedings of the 5th ACM Workshop on Wireless Security*, New York, NY, USA, 2006.

[19] Y. Yang, X. Wang, S. Zhu and G. Cao, "Distributed software-based attestation for node compromise detection in sensor networks," in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, Washington, DC, USA, 2007.

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

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

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

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

[24] D. Schellekens, B. Wyseur and B. Preneel, "Remote Attestation on Legacy Operating Systems with Trusted Platform Modules," *Sci. Comput. Program,* vol. 74, no. no. 1-2, p. 13–22, Dec 2008.

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

[26] P. England, B. Lampson, J. Manferdelli, M. Peinado and B. Willman, "A Trusted Open Platform," *Computer,* vol. 36, no. no. 7, p. 55–62, Jul 2003.

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

[28] P. England, "Practical techniques for operating system attestation," in *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, Berlin, Heidelberg, 2008.

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

[30] A. Resh and N. Zaidenberg, "Can keys be hidden inside the CPU on modern windows host," in *ECIW*, Jyvaskyla, 2013.

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

[32] M. Kiperberg and N. Zaidenberg, "Efficient Remote authentication," *Journal of Information warfare,* 2013.

[33] M. Howard, M. Miller, J. Lambert and M. Thomlinson, "Windows isv software security defenses," Microsoft Corporation, 2010. [Online]. Available: https://msdn.microsoft.com/ en-us/library/bb430720.aspx.

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

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

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

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