Hypervisor-based Whitelisting of Executables

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

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

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

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

I. Introduction

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

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

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

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

Our method can be categorized as whitelist-based as it permits the creation of a whitelist database of allowed exe-

cutables that will be used by the system to enforce authorized execution. However, our method does not suffer from two main deficiencies present in current methods:

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

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

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

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

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

A. VMX

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

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

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

B. System description

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

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

II. Preparations

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

A. User mode scanning

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

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

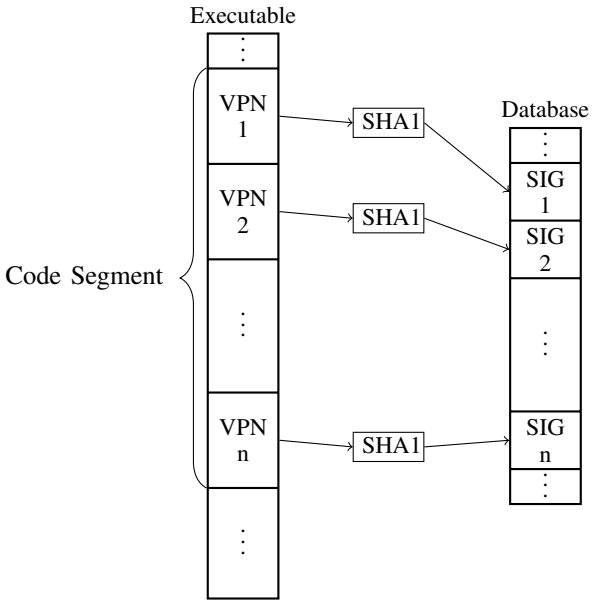

Fig. 1. The user mode executable image to be signed (left) is composed of one code segment, divided to virtual-pages. Each page is signed using SHA1 and the result is stored in the whitelist database (right).

B. Kernel mode scanning

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

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

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

1) During its initialization, the Linux kernel may optionally mount an initial ramdisk (*initramfs*), if found. The purpose of the initial ramdisk is to mount the root file system. In many Linux distributions (mainly general distributions), the initramfs also includes kernel modules because the machine on which the kernel will run is not known in advance. We built the latest stable Linux kernel to date (4.15.10, 19.03.2018) on a random machine, having all the necessary kernel modules statically compiled into the kernel. The latter can be achieved using 'localyesconfig' make target. Because the necessary kernel modules are statically compiled into the kernel, it was not necessary to boot the system with an initial ramdisk.

2) We booted up the system (Ubuntu 16.04.4 LTS) with the just-built kernel and disabled kaslr. Afterwards, we scanned the kernel directly from the main memory. Because the executable scanner cannot directly access kernel space memory (as it executes in user mode), the hypervisor provides a hypercall service that can be used to compute a hash of a given kernel page. The executable scanner uses the latter service to generate hashes of all active kernel-pages. These hashes are written directly into the whitelist database consecutively (just as in user mode applications).

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

C. Configuring secure boot

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

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

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

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

database of executable images. Updates to the *db* and *dbx* variables need to be signed using one of the keys within the *kek* variable.

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

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

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

Steps for option 2:

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

D. Target installation

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

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

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

III. Operation

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

A. Initialization

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

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

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

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

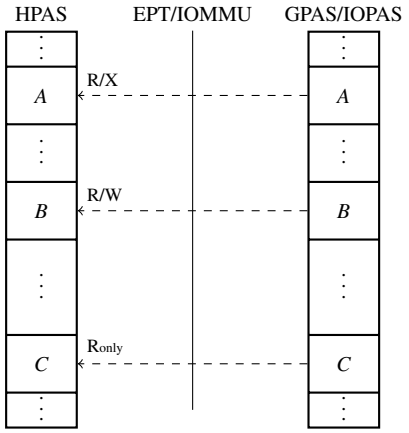

Fig. 2. Physical address space as perceived by the guest and I/O peripherals (right). Physical page *A* contains code that was previously authenticated by the hypervisor. Therefore, it has read/execute rights. Physical page *B* has yet to be executed. Therefore, it has read/write rights. Physical page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

B. System monitoring

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

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

C. OS kernel monitoring

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

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

IV. Security

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

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

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

A. HV memory protection

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

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

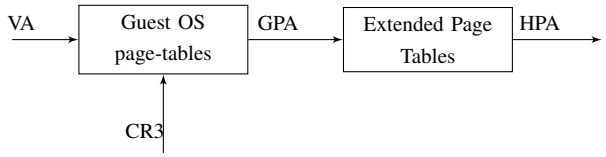

Fig. 3. Address translation process of a guest in Intel processors. First, the guest-linear address is translated into a guest-physical-address by the OS page-tables. Second, the guest-physical-address is translated into a host-physical-address by the hypervisor's extended-page-tables

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

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

B. Secure boot

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

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

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

V. Performance

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

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

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

All experiments were performed in the following environment:

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

A. Page verification forcing experiment

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

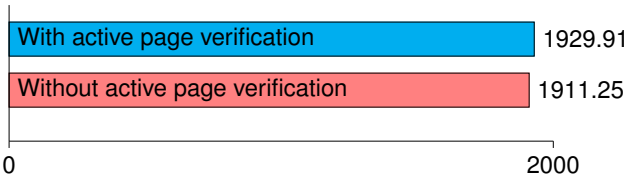

Fig. 4. Thousands of cycles (less is better) for a single call and return.

B. Empirical evaluation

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

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

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

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

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

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

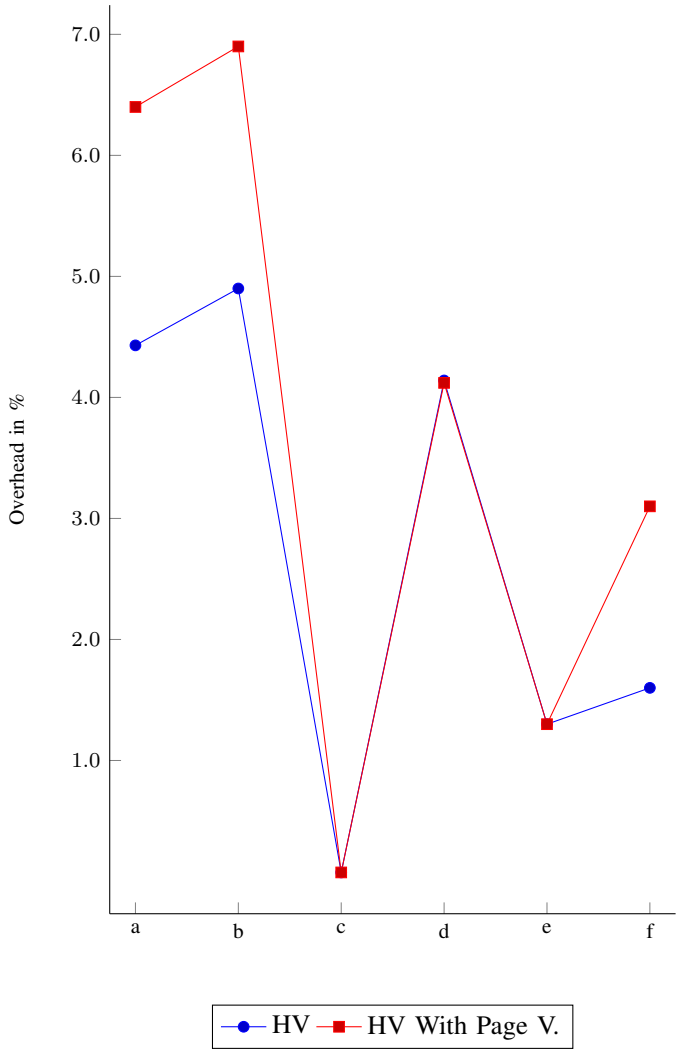

Fig. 5. Overhead of the benchmark execution under two conditions: (a) with HV only, and (b) with HV and enabled page verification

VI. Limitations and Future Work

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

A. SMP support

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

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

B. Other OSes support

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

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

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

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

C. Managed code

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

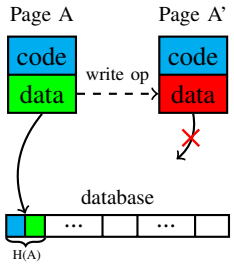

Fig. 6. Page *A* is a partial code page (i.e., contains both code and data). Initially, the data part has not yet been modified; therefore, *H(A)* is located within the whitelist database. Later, the data part of page *A* is modified. Even though the code is left untouched, an execution attempt of *A'*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

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

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

D. Control flow and data integrity

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

E. ARM architecture

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

F. Other applications

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

VII. Conclusions

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

References

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

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

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

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

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

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

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