Hypervisor-based Protection of Code

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

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

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

I. Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

A. VMX

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

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

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

B. System Description

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

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

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

C. Threat Model

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

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

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

We admit that the system is vulnerable to attacks that:

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

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

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

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

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

II. Related Work

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

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

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

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

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

III. The Hypervisor

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

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

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

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

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

IV. Preparations

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

A. Encryption

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

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

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

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

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

B. Target Installation

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

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

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

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

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

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

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

V. Operation

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

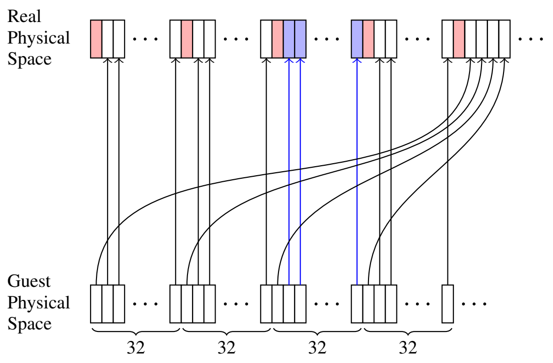

Fig. 1. EPT configuration that maps guest physical pages (bottom) to real physical pages (top). Real physical pages whose index is a multiple of 32 (red rectangles) are safe pages, which are used exclusively by the hypervisor. Guest's physical pages whose index is a multiple of 32 are mapped to the last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

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

A. Initialization

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

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

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

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

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

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

B. Transitions

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

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

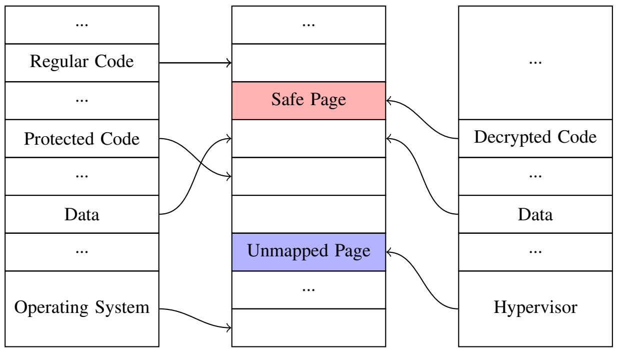

Fig. 2. Virtual address space layouts of the hypervisor and the guest during protected function execution. The code and the data structures of the hypervisor are not mapped in the guest. The protected code is decrypted to a safe page. The virtual address of the protected function in the hypervisor corresponds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

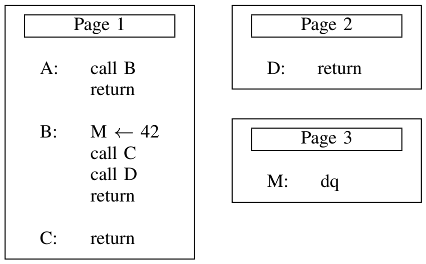

Fig. 3. Sample program. The program consists of four functions (A, B, C, D) and of one variable (M). The functions A, B and C reside in the first page. The function D resides in the second page. The variable M resides in the third page. In this program only the function B is encrypted.

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

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

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

Algorithm 1 Hypervisor's VM-exit handler

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

The hypervisor reacts to a special instruction-induced general protection exception by entering a loop (line 4–16). At the beginning of each iteration (line 5), the hypervisor transitions to user-mode (without returning to guest) and sets the instruction pointer to the same address that generated the general protection exception. The execution continues until an exception occurs in user-mode. We demonstrate the handling of different exceptions by an example (see Fig. 3), and then describe the algorithm line-by-line.

Consider a program that contains 4 functions A, B, C, D and a variable M, that are stored in three memory pages: the first page contains the functions A, B and C, the second page contains the function D, and the third page contains the variable M. The functions are implemented as follows: A calls B and returns, C and D return immediately, B accesses the variable M, calls C, calls D and returns. We now dissect the execution of this program under the assumption that the function B is encrypted and that the page containing M is not mapped in the operating system.

1) The function A executes normally and calls B. The function B attempts to execute an `HLT`. A VM-exit

occurs and the hypervisor enters user-mode (line 5). Since the page containing B is not mapped in the hypervisor, a page-fault occurs. The error code of this page-fault indicates that it was caused by an instruction fetch (denoted by #PF[INSTR] on line 6). The instruction pointer is in a protected function (the condition on line 7 does not hold) but the address is not mapped in the hypervisor (the condition on line 9 does hold), therefore the hypervisor allocates a *safe page* (which is a physical page), fills it with `HLT`s, and maps it to the (virtual) page containing B. Finally, the hypervisor locates the entry corresponding to B in the *database*, and decrypts it.

2) The loop continues. The hypervisor enters user-mode. The function B accesses the variable M, which is not mapped in the hypervisor. A page-fault exception occurs. The error code indicates data access (denoted by #PF[DATA] on line 13). According to our assumption, the page that contains the variable M is not mapped in the guest. Therefore, the hypervisor injects the exception to the guest (line 16).
3) The guest operating system maps the page that contains the variable M, and resumes the execution of the function B. However, from the guest's perspective B still contains `HLT`s, and its execution causes a VM-exit.
4) The hypervisor enters user-mode (line 5), which immediately generates a page-fault exception, since the page that contains the variable M is still not mapped in the hypervisor. The hypervisor copies the mapping from the guest (line 14) and the loop continues.
5) The hypervisor enters user-mode (line 5). The function B continues its execution and calls the function C, which resides with B in the same page. This page was filled with `HLT`s during its allocation. A general-protection exception occurs (line 6). Since the instruction pointer is not in a protected function (line 7), the hypervisor returns to the guest.
6) The function C executes as usual in the guest and then returns to B, which is filled with `HLT`s from the perspective of the guest. A VM-exit occurs. The hypervisor enters user-mode and the execution of B continues, until it calls D. Since the page containing D is not mapped, a page-fault exception occurs. The error code indicates instruction fetch (denoted by #PF[INSTR] on line 6). The instruction pointer is not in a protected function (the condition on line 7 holds), therefore the hypervisor returns to the guest.
7) The function D executes as usual in the guest and then returns to B. A VM-exit occurs. The hypervisor enters user-mode (line 5). The function B continues and eventually returns to A. From the hypervisor's perspective the function A is filled with `HLT`s. A general-protection exception occurs. Since the instruction pointer is not in a protected function, the hypervisor returns to the guest (line 8).

The algorithm begins by checking the reason for the VM-exit. If the VM-exit is due to a general-protection exception that was not caused by a `HLT`, the hypervisor injects this exception to the guest (line 2). If the VM-exit is due to execution of a protected function, the hypervisor begins a loop. At the beginning of each iteration, the hypervisor attempts to execute the protected function. Eventually the execution generates an exception. The hypervisor checks the type of the generated exception.

- If it is either a general-protection exception, which is generated by jumping to an unknown location (which does not contain a decrypted function) in a *safe page*, or a page-fault exception caused by an instruction fetch, the hypervisor checks whether the instruction pointer resides in a protected function. If the instruction pointer is not in a protected function, the hypervisor returns to the guest (line 8), in order to continue the execution there. If the instruction pointer is in a protected function, it is possible that the page containing this function is not yet mapped in the hypervisor. In such case, the hypervisor allocates a *safe page* and fills it with `HLT`s. The filling guarantees that any jump outside a decrypted function generates an exception. Finally, the hypervisor decrypts the protected function.
- If the exception type is a page-fault caused by data access and this data is mapped in the guest, then the hypervisor copies the mapping from the guest (line 14). If, however, the data is not mapped in the guest, then the hypervisor injects the exception to the guest, thus requesting the operating system to map the accessed data.

During execution of regular functions or handling of interrupts and exceptions, the operating system may modify the mappings of virtual pages. The hypervisor may have copies of some of these mappings, which were modified by the operating system. Thus, it is essential for the hypervisor to intercept all such modifications. Fortunately, according to Intel's specification [32], since the processor stores portions of mapping information in its caches (TLBs), the operating system is required to inform the processor of all modifications through a special instruction, `INVLPG`. The hypervisor intercepts this instruction, and responds to it by erasing all the entries that were copied from the operating system (lines 17–19).

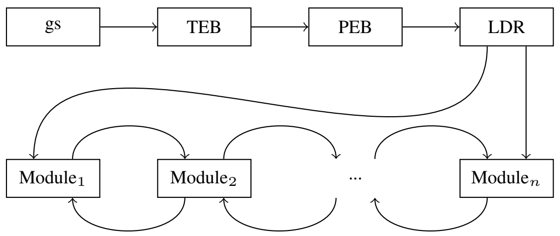

Fig. 4. The path to the executable images of the current process. The special register ”gs” points to a data structure that describes the current thread (TEB). The TEB data structure points to a data structure that describes the current process (PEB). The PEB data structure points to a data structure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

C. OS Dependence

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

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

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

VI. Security

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

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

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

We admit that the system is vulnerable to attacks that:

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

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

A. Memory Protection

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

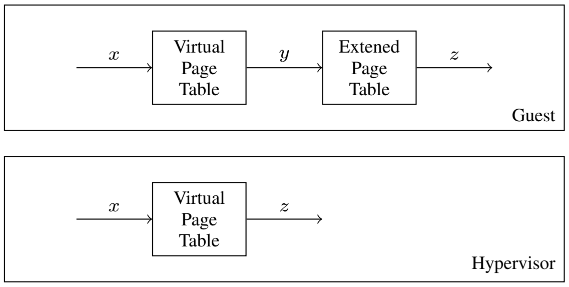

Fig. 5. Address translations in the guest and the hypervisor. When the guest accesses a virtual address x, it is translated to a guest physical address y via the guest's virtual page table, then y is translated to a real physical address z via EPT. When the hypervisor accesses a virtual address x, it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

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

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

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

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

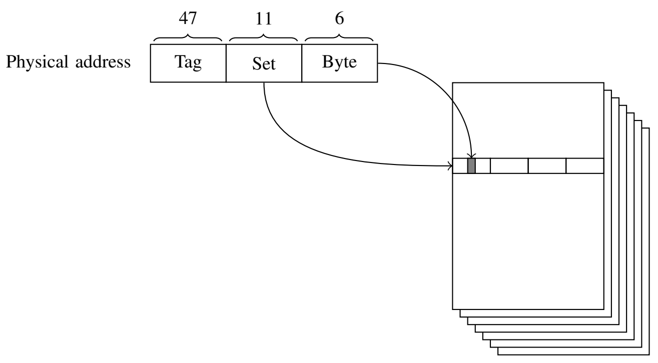

Fig. 6. An example of a last level cache organization of associativity 4 with 6 slices and 2048 sets in each slice. The 6 least significant bits of the physical address select the byte in the cache line. The next 11 bytes select the set. The slice is determined by applying a hash function to the set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

B. Cache Evictions

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

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

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

C. Decryption Key

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

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

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

Algorithm 2 UEFI Application Initialization Sequence

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

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

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

VII. Performance

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

The evaluation presented below answers the following questions:

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

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

All the experiments were performed in the following environment:

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

A. Hypervisor Performance Impact

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

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

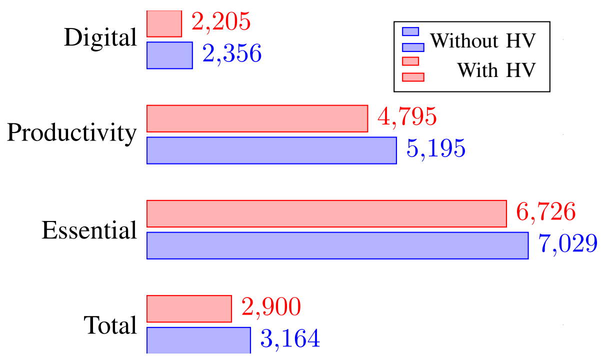

Fig. 7. The scores (larger is better) reported by PCMark in 4 categories: Digital Content Creation, Productivity, Essential and Total.

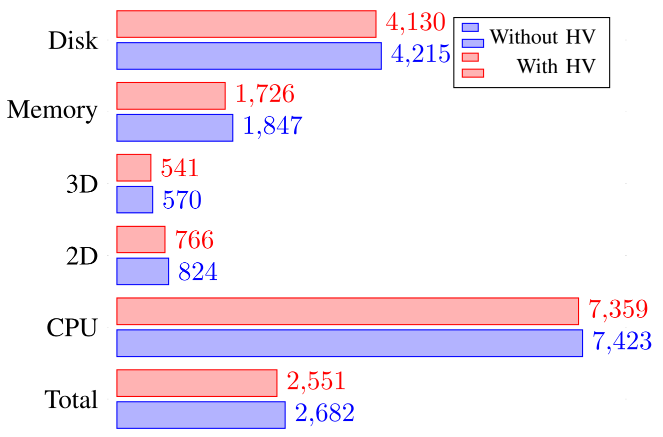

Fig. 8. The scores (larger is better) reported by PassMark in 6 categories: Disk, Memory, 3D, 2D, CPU and Total.

3) Novabench. Version Info: 4.0.3 November 2017.

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

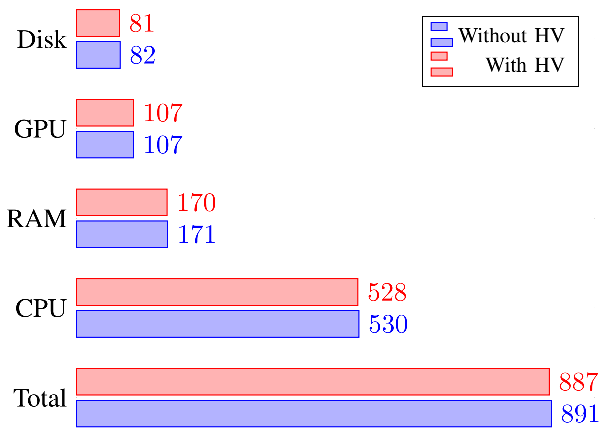

Fig. 9. The scores (larger is better) reported by Novabench in 5 categories: Disk, GPU, RAM, CPU and Total.

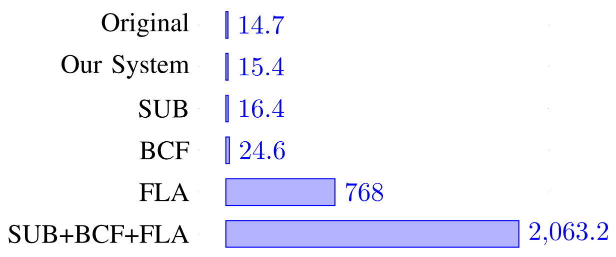

Fig. 10. Execution times in seconds of the original, encrypted and obfuscated versions 7-Zip in the second experiment.

B. Comparison

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

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

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

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

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

C. Randomized Experiment: Lame

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

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

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

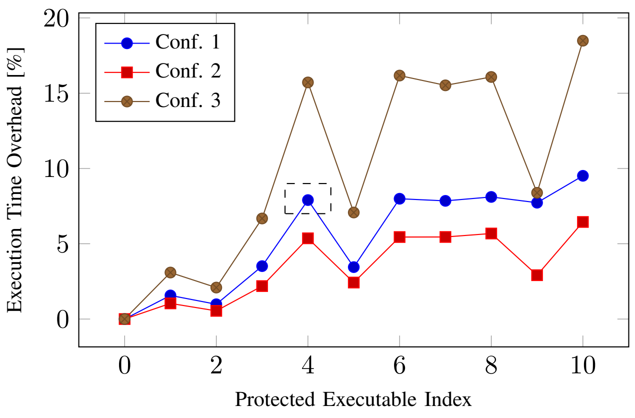

Fig. 11. The overhead of protected executables. Each line represents a single configuration. A mark on a line represents an average execution time overhead (in percents) of a protected executable compared to the original executable. Consider the mark inside the dashed square, which corresponds to P_4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

D. Randomized Experiment: 7-Zip

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

E. Built-in Profiler

The built-in profiler allows one to get a better view of the relationships between the encrypted and non-encrypted functions. This information might be critical as every branching

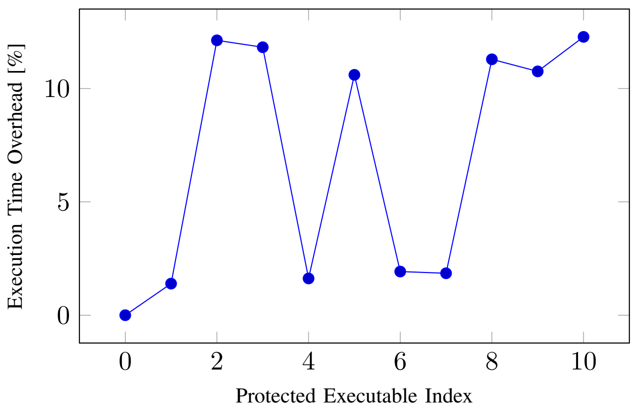

Fig. 12. The overhead of protected executables. A mark on a line represents an average execution time overhead (in percents) of a protected executable compared to the original executable.

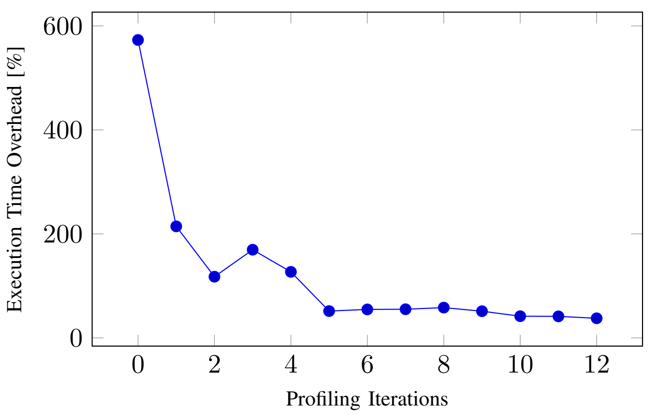

Fig. 13. The overhead of the protected executable during profiling iterations. A mark at position i represents the overhead in percents of the protected executable compared to the original executable at the ith iteration. Note the exponential behavior of the overhead improvement.

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

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

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

F. Case Study: OpenSSL

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

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

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

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

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

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

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

We performed 12 iterations in total. The last profiling iteration was composed of 59 encrypted functions from which 61% are a result of type 1 branching and 39% are a result of type 2 branching. During our tests we found that encrypting type 3 branching until our last phase gave little benefit. The execution time overhead of the last iteration was 18% of the original's. It should be noted that the process could continue further in order to achieve even better results. Fig. 13 depicts the results we obtained.

	Compression	Decompression
Original	122.7143	6.4979
Protected	122.7701	6.5617

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

G. Case Study: 7-Zip

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

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

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

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

H. Case Study: Apache Web Server

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

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

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

VIII. CONCLUSIONS

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

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

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

REFERENCES

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

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

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

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

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

[6] S. Bhatkar and R. Sekar, “Data space randomization,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 1–22.

[7] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, “Data randomization,” Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, Tech. Rep., 2008.

[8] S. Forrest, A. Somayaji, and D. H. Ackley, “Building diverse computer systems,” in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE, 1997, pp. 67–72.

[9] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Smashing the gadgets: Hindering return-oriented programming using in-place code randomization,” in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[10] R. El-Khalil and A. D. Keromytis, “Hydan: Hiding information in program binaries,” in *International Conference on Information and Communications Security*. Springer, 2004, pp. 187–199.

[11] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, “The superdiversifier: Peephole individualization for software protection,” in *International Workshop on Security*. Springer, 2008, pp. 100–120.

[12] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, “Code obfuscation against symbolic execution attacks,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 189–200.

[13] A. Salem and S. Banescu, “Metadata recovery from obfuscated programs using machine learning,” in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. ACM, 2016, p. 1.

[14] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, “A generic approach to automatic deobfuscation of executable code,” in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 674–691.

[15] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 25–36.

[16] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. De Bosschere, "Towards tamper resistant code encryption: Practice and experience," in *International Conference on Information Security Practice and Experience*. Springer, 2008, pp. 86–100.

[17] J. Bringer, H. Chabanne, and E. Dottax, "White box cryptography: Another attempt." *IACR Cryptology ePrint Archive*, vol. 2006, no. 2006, p. 468, 2006.

[18] M. Karroumi, "Protecting white-box aes with dual ciphers," in *International Conference on Information Security and Cryptology*. Springer, 2010, pp. 278–291.

[19] Y. Xiao and X. Lai, "A secure implementation of white-box aes," in *Computer Science and its Applications, 2009. CSA'09. 2nd International Conference on*. IEEE, 2009, pp. 1–6.

[20] O. Billet, H. Gilbert, and C. Ech-Chatbi, "Cryptanalysis of a white box aes implementation," in *International Workshop on Selected Areas in Cryptography*. Springer, 2004, pp. 227–240.

[21] Y. De Mulder, P. Roelse, and B. Preneel, "Cryptanalysis of the xiao–lai white-box aes implementation," in *International Conference on Selected Areas in Cryptography*. Springer, 2012, pp. 34–49.

[22] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel, "Cryptanalysis of white-box des implementations with arbitrary external encodings," in *International Workshop on Selected Areas in Cryptography*. Springer, 2007, pp. 264–277.

[23] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, and B. De Sutter, "Tightly-coupled self-debugging software protection," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. ACM, 2016, p. 7.

[24] P. Ferrie, "Attacks on more virtual machine emulators," *Symantec Technology Exchange*, vol. 55, 2007.

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

[26] J. Kinder, "Towards static analysis of virtualization-obfuscated binaries," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 61–70.

[27] R. Rolles, "Unpacking Virtualization Obfuscators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1.

[28] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[29] "Stunnix C/C++ Obfuscator," http://stunnix.com/, 2018, [Online; accessed 25-Feb-2018].

[30] "The Tigress C Diversifier/Obfuscator," http://tigress.cs.arizona.edu/, 2018, [Online; accessed 25-Feb-2018].

[31] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium (USENIX Security), Washington, DC*, 2015.

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

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

[34] "ARM architecture reference manual ARMv8-A," ARM Ltd., 2013.

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

[36] M. Lipow, "Number of faults per line of code," *IEEE Transactions on software Engineering*, no. 4, pp. 437–439, 1982.

[37] O. Alhazmi, Y. Malaiya, and I. Ray, "Security vulnerabilities in software systems: A quantitative perspective," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2005, pp. 281–294.

[38] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," *CanSecWest/core06*, 2006.

[39] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 335–350.

[40] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 380–395.

[41] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "Hima: A hypervisor-based integrity measurement agent," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 461–470.

[42] J. Rutkowska and R. Wojtczuk, "Qubes os architecture," *Invisible Things Lab Tech Rep*, vol. 54, 2010.

[43] S. T. King and P. M. Chen, "Subvirt: Implementing malware with virtual machines," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 14–pp.

[44] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[45] PELock LLC. PELock: Software Protection. [Online]. Available: https://www.pelock.com/

[46] The UPX Team. UPX: the Ultimate Packer for eXecutables. [Online]. Available: https://upx.github.io/

[47] J. Raber, "Columbo: High perfomance unpacking," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 507–510.

[48] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177.

[49] VMware. VMware Workstation Pro. [Online]. Available: https://www.vmware.com/il/products/workstation-pro.html

[50] Oracle. VirtualBox. [Online]. Available: https://www.virtualbox.org/

[51] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 143–158.

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

[53] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Science of Computer Programming*, vol. 74, no. 1-2, pp. 13–22, 2008.

[54] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[55] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. OHanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.

[56] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[57] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 338–359, 1992.

[58] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 89–102.

[59] Intel. 8th Generation Intel Core i7 Processors. [Online]. Available: https://ark.intel.com/products/series/122593/8th-Generation-Intel-Core-i7-Processors

[60] C. Lambrinoudakis, G. Pernul, and M. Tjoa, *Trust, Privacy and Security in Digital Business: 4th International Conference, TrustBus 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007. [Online]. Available: https://books.google.co.il/books?id=ci-rNuWTLa4C

[61] "LAME," http://lame.sourceforge.net/, 2018, [Online; accessed 25-Feb-2018].

[62] OpenSSL Software Foundation, "OpenSSL: Cryptography and SSL/TLS Toolkit," https://www.openssl.org/, 2018, [Online; accessed 25-Feb-2018].

[63] Apache. The Apache HTTP Server Project. [Online]. Available: https://httpd.apache.org/

[64] V. Costan and S. Devadas, "Intel SGX Explained." *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.