System for Executing Encrypted Native Programs

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

Abstract

An important aspect of protecting software from attack, theft of algorithms, or illegal software use, is eliminating the possibility of performing reverse engineering. One common method to deal with these issues is code obfuscation. However, in most case it was shown to be ineffective. Code encryption is a much more effective means of defying reverse engineering, but it requires managing a secret key available to none but the permissible users. The authors propose a new and innovative solution. Critical functions in protected software are encrypted using well-known encryption algorithms. Following verification by external attestation, a thin hypervisor is used as the basis of an eco-system that manages just-in-time decryption, inside the CPU, where decrypted instructions are then executed and finally discarded, while keeping the secret key and the decrypted instructions absolutely safe. The paper presents and compares two methodologies that perform just-in-time decryption: in-place and buffered execution. The former being safer, while the latter boasts better performance.

Keywords*: Hypervisor, Trusted computing, Attestation, Cyber-security*

1. Introduction

Digital content such as games, videos, and the like may be susceptible to unlicensed usage, which has a significant adverse impact on the profitability and commercial viability of such products. Commonly, such commercial digital content may be protected by a licensing verification program; these, however, may be circumvented by reverse engineering of the software instructions of the computer program which leaves them vulnerable to misuse.

One way of preventing circumvention of the software licensing program, may be using a method of obfuscation [1] [2]. The term obfuscation refers to making software instructions difficult for humans, as well as reverse-engineering software tools, to understand by deliberately cluttering the code with useless, confusing pieces of additional software syntax or instructions. However, even when changing software code and making it obfuscated, the content is still readable to the skilled hacker [3] [4].

Additionally, publishers may protect their digital content product by encryption, using a unique key to convert the software code to an unreadable format, such that only the owner of the unique key may decrypt the software code. Such protection may only be effective when the unique key is kept secured and unreachable to an adversary. Hardware based methods for keeping the unique key secured are possible [5] [6] [7], but may have significant deficiencies, mainly due to an investment required in dedicated hardware on the user side, making it costly, and, therefore, impractical. Furthermore, such hardware methods have been successfully attacked by hackers [8] [9].

Software copy-protection is currently predominantly governed by methodologies based on obfuscation, which are volatile to hacking or user malicious activities. There is, therefore, a need for a better technique for protecting sensitive software sections, such as licensing code.

In this paper, we present a system that allows encrypting and executing native programs written for the x86 architecture. The system is based on the approach proposed by Averbuch et al. [10], in which an attested kernel module is responsible for decryption and execution of encrypted functions. The main deficiency of the proposed approach is the inability of the kernel module to protect itself from the operating system. As a consequence, a vulnerability in the operating system may compromise the secret key. Moreover, the attestation server has to attest not only the kernel module responsible for decryption but also the entire operating system. The complications of operating system attestation and a partial mitigation are described in [11].

This paper proposes to solve all these complications by utilizing the virtualization extension, which is available on modern processors [12] [13], in order to enable the decrypting kernel module to protect itself, thus eliminating the need for operating system attestation. Figure 1 depicts the components of the proposed system as well as their relationships. The system is deployed on three computers: a development machine, on which the program to be encrypted, is compiled and encrypted; the attestation server, which stores the decryption key, and delivers it to the target machine; and the target machine, which executes the encrypted program. A special driver, which embeds a hypervisor, is installed on the target machine prior to execution of an encrypted program. The hypervisor obtains the decryption key, which is necessary for program execution, from the attestation server, when an encrypted program is loaded to the memory.

1.1 Intel SGX

Intel has announced its new security technology named Software Guard Extensions (SGX) [32], which enables developers to create secure containers, called enclaves, inside a process address space. The enclave address space is protected from any other software not resident in the enclave, including privileged software. This guarantees that malware, at any privilege level, cannot compromise the confidentiality or integrity of enclave resident software or data. SGX does not rely on a hypervisor or hardware virtualization, instead it encompasses two new instruction-set extensions that allow initializing and managing the enclaves. Secure storage is managed in an Enclave-Page-Cache, which is protected by hardware from "non-enclave" access. SGX provides the means for implementations to the same end as proposed by our methodology, however the SGX processor extensions are available only in the newest Intel processors. Therefore, utilizing an SGX based solution requires specific hardware, adds to equipment cost and is not supported on legacy systems.

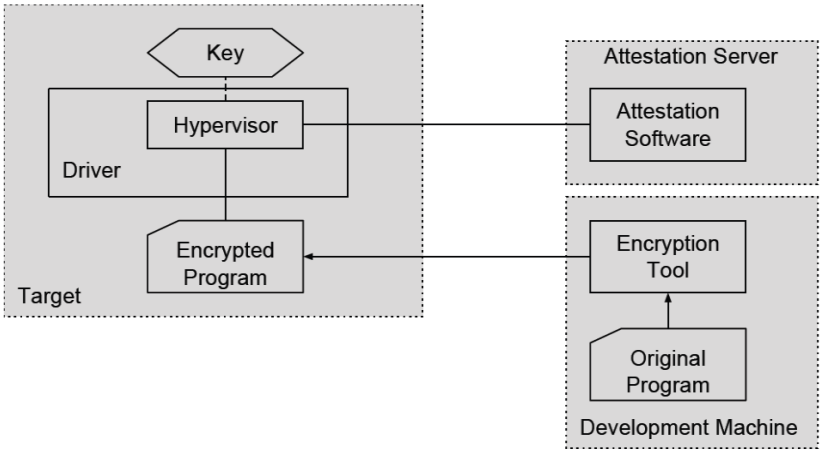

Figure 1. Native code protection system. The original program is encrypted before its distribution. The encryption key is stored in the attestation server, which delivers it to the hypervisor in the target machine upon successful attestation. The hypervisor is initialized by a driver, which also hosts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

1.2 Contribution

The methodology proposed in this paper provides for a software-only solution, based on the availability of hardware virtualization and secondary-level address translation, incorporated in most Intel and AMD CPUs released after 2008. Furthermore, an innovative thin hypervisor is utilized to protect cryptographic keys and decrypted code to provide a truly secure just-in-time code decryption mechanism. The thin hypervisor is guaranteed to be trusted with the employment of remote attestation.

2. Encryption tool

The encryption tool is responsible for encryption of selected functions in a program. The user selects the functions to be encrypted by specifying their names in a configuration file. A *map file* or a *debug symbols file*, which are produced by a compiler, can then be used to translate the names of the functions to their locations in the program file.

On Windows, program files, executables and dynamic libraries, are stored in Portable Executable (PE) format [14]. Figure 2 depicts the structure of a PE file. The different headers define the expected location of the PE file when loaded to memory, sizes and positions of various data structures inside the PE file, the number of sections contained in this PE file, etc. The section table contains a description of each of the sections contained in the PE file. Following the section table are the sections themselves. Sections vary in their structure and purpose: the .text section contains the code of the program, the .data section contains its constants. Other sections may contain information about resources (images and sounds) embedded in the PE file or information used during exception delivery.

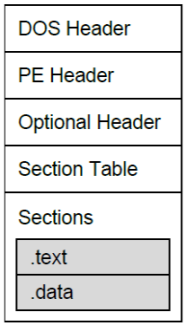

Figure 2. Structure of a Windows PE file. The structure contains a variable number of sections. Two of the most common sections are presented.

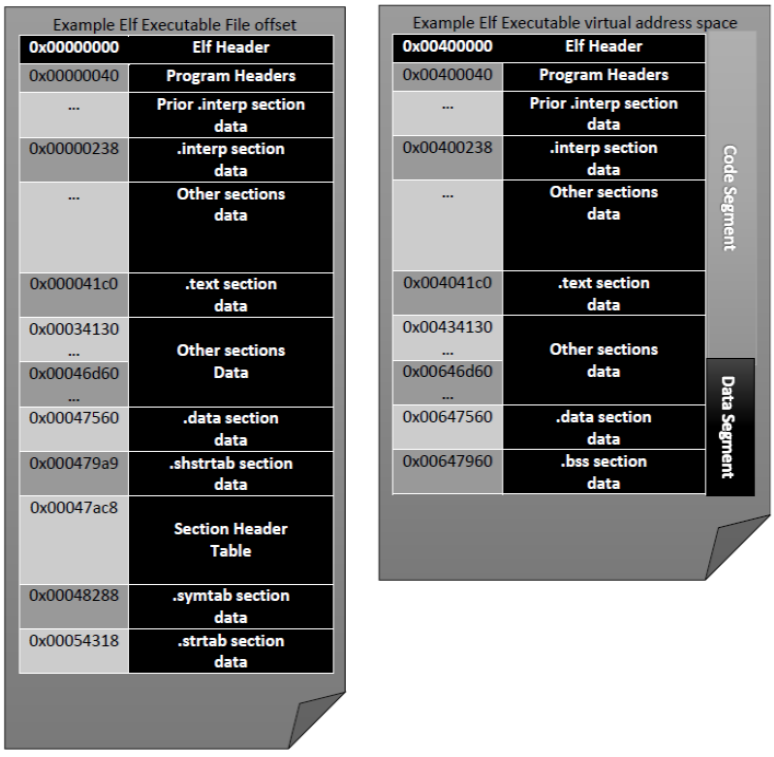

Figure 3. The left image represents the structure of an ELF file as it is stored in disk. The right image represents the structure of an ELF file as it is loaded to memory.

On Linux, program files, executable files and dynamic libraries, are stored in Executable and Linkable Format (ELF) format [15]. Figure 3 depicts the structure of an ELF file. An ELF file consists of a header, which is followed by data. The data may include:

- Program header table, describing zero or more segments. Only two segments can be defined as loadable: the code segment and the data segment. The code segment is loaded to memory with read-write-execute permissions, while the data segment is loaded with read-only permissions. Other segments are not loaded to memory.
- Section header table, describing zero or more sections. A typical ELF file holds a section called *.text*, which contains the code of the program.
- Data referenced by entries in the program header table or section header table.

The segments contain information that is necessary for runtime execution of the file, while the sections contain data for linking and relocation. Figure 3 depicts the structure of an ELF virtual-image at load time.

The encryption tool modifies the given PE/ELF file by introducing a new section, which stores the selected functions in encrypted form. The instructions of the original functions are partially replaced by an exception inducing instruction. We propose to use either the *halt* instruction or the *software breakpoint* instruction. The halt instruction is a privileged instruction, which deactivates the current processor when executed in kernel mode, but generates a general protection fault when executed in user mode. The software breakpoint instruction generates a breakpoint trap when executed in either kernel or user modes. Faults and traps, being types of interrupts, can be intercepted by a hypervisor, which can then decrypt and execute the original encrypted function. Another benefit of the halt and the software breakpoint instructions is that they can be represented by a single byte (0xF4 for halt and 0xCC for software breakpoint), thus allowing them to fully cover any number of bytes. The software breakpoint instruction is superior to the halt instruction in that it generates an interrupt not only in user mode but also in kernel mode.

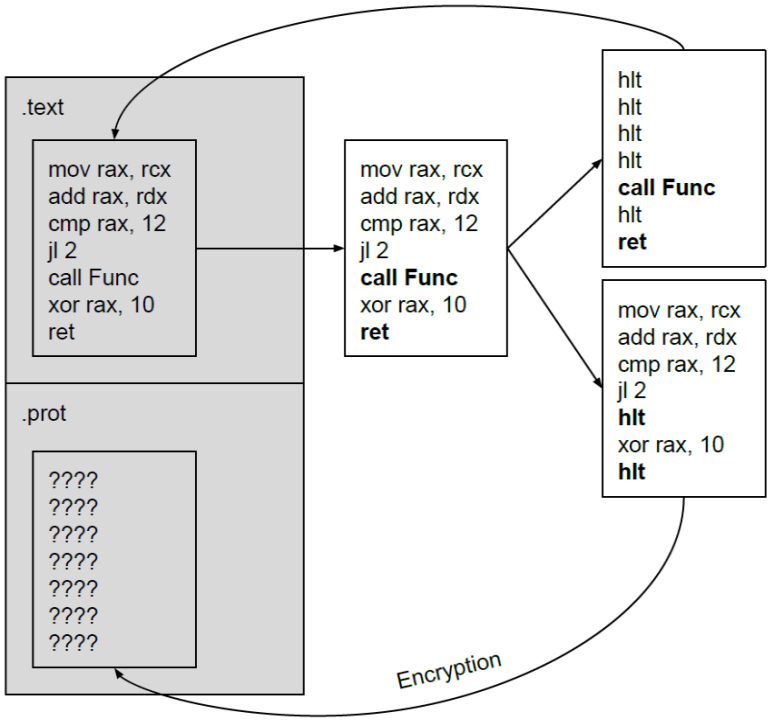

Figure 4. Example of an encryption process of a single function. The encryption begins by classifying instruction is encryptable (normal face) and non-encryptable (bold face), and creating to copies. The complementary instructions in each copy are replaced by halts. Finally, one copy is written over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

As will be explained in section 5, it is highly important to intercept control transfers that leave the encrypted function. The encryption tool disassembles the function to be encrypted and inspects its instructions. The instructions then are classified as *encryptable* and *non-encryptable*. The encryption tool classifies an instruction as non-encryptable if it might transfer control out of the encrypted function. For example, the ret and the call instructions are always classified as non-encryptable, but the jmp instruction is classified as non-encryptable only if its destination lays outside of the protected function's bounds or if the destination cannot be determined statically (if it is stored in a register, for instance).

The encryption tool produces two copies of the original function, the encryptable copy (EC) and the non-encryptable copy (NEC). In the EC all the non-encryptable instructions are replaced by the halt or the software breakpoint instructions. Then the encryption tool encrypts the EC and stores it in the new section. In the NEC all the encryptable instructions are replaced by the halt or the software breakpoint instructions. Then the encryption tool replaces the original function by the NEC. Figure 4 presents an example of such a transformation.

3. Hypervisor

A hypervisor, also referred to as a Virtual Machine Monitor (VMM), is software, which may be hardware-assisted, to manage multiple virtual machines on a single system [16]. The hypervisor virtualizes the hardware environment in a way that allows several virtual machines, running under its supervision, to operate in parallel over the same physical hardware platform, without obstructing or impeding each other. Each virtual machine has the illusion that it is running unaccompanied on the entire hardware platform. The hypervisor is referred to as the *host*, while the virtual machines are referred to as *guests*.

A virtual machine control structure (VMCS) is defined for each virtual environment managed by a virtual machine monitor (VMM) [12]. This structure defines the values of privileged registers, the location of the interrupt descriptors table, and additional values that constitute the internal state of the virtual environment. In addition, this structure defines the events that the VMM is configured to intercept, and the address of the function that should handle the interception. The act of control transfer from the virtual environment to a predefined function is called vm-exit and the act of control transfer from the function back to the virtual environment is called vm-entry. Upon vm-exit the function can determine the reason of the vm-exit by examining the fields of the VMCS and altering them, thus altering the state of the virtual environment as it wishes. Finally, the VMCS can define a mapping between the physical memory as it is perceived by the virtual environment and the actual physical memory. As a consequence, the VMM can prevent access to some physical pages by the virtual environment. Moreover, the virtual environment will be unaware of this situation.

We propose to use a hypervisor for securing a single guest. Rather than wholly virtualizing the hardware platform, a special breed of hypervisor, called a *thin hypervisor*, is used [17] [18]. A thin hypervisor is configured to intercept only a small portion of events. All other events are processed without interception, directly, by the OS. A thin hypervisor only intercepts the set of events that allows it to protect an internal secret (such as a cryptographic key) and protect itself from subversion. Figure 5 depicts a thin hypervisor supporting a single guest. Since a thin hypervisor does not control most of the OS interaction with the hardware, multiple OS are not supported. On the other hand, system performance is kept at an optimum.

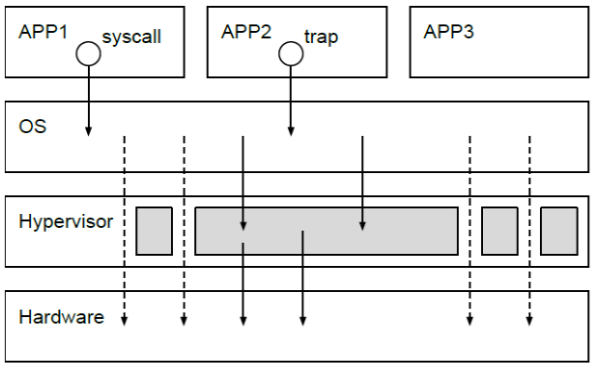

Figure 5. Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions, and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service from the underlying hardware. A thin hypervisor can intercept some of those requests and handle them according to some policy.

A thin hypervisor facilitates a secure environment by: (a) setting aside portions of memory that cannot be accessed by the guest, (b) storing the cryptographic key in privileged registers, and (c) intercepting privileged instructions that may compromise its protected memory, reveal the cryptographic key, or attempt to subvert the hypervisor.

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

4. Remote attestation

The problem of remote software authentication, determining whether a remote computer system is running the correct version of a software, is well known [5] [19-25][33]. Equipped with a remote authentication method, a service provider can prevent an unauthenticated remote software from obtaining some secret information or some privileged service. For example, only authenticated gaming consoles can be allowed to connect to the gaming networks [26-28], and only authenticated bank terminals can be allowed to fetch records from the bank database [29]. We have also shown that once attestation is completed the attested computer can receive encryption keys from the attestation server and protect them from malicious software in a modern host [34].

The research in this area can be divided into two major branches: hardware assisted authentication [5-7] and software-only authentication [19-22]. In this paper we concentrate on software-only authentication, although the system can be adapted to other authentication methods, as well. The authentication entails simultaneously authenticating some software component(s) or memory region, as well as verifying that the remote machine is not running in virtual or emulation mode. Software-only authentication methods may also involve a challenge code that is sent by the authentication authority, and executed on the remote system. The challenge code computes a result that is then transmitted back to the authority. The authority deems the entity to be authenticated if the result is correct and was received within a predefined time-frame. The underlying assumption, which is shared by all such authentication methods, is that only an authentic system can compute the correct result within the predefined time-frame. The methods differ in the means by which (and if) they satisfy this underlying assumption.

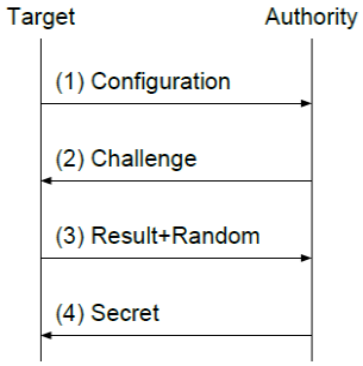

Figure 6. The attestation protocol between the authentication authority and the target machine. The protocol consists of four messages. The first two messages are sent unencrypted, while the two last messages are encrypted. The third message is encrypted by the public key of the authentication authority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

Kennell and Jamieson proposed [19] a method that produces the result by computing a cryptographic hash of a specified memory region. Any computation on a complex instruction set architecture (Pentium in this case) produces side effects. These side effects are incorporated into the result after each iteration of the hashing function. Therefore, an adversary, trying to compute the correct result on a non-authentic system, would be forced to build a complete emulator for the instruction set architecture to compute the correct side effects of every instruction. Since such an emulator performs tens and hundreds of native instructions for every simulated instruction, Kennell and Jamieson conclude that it will not be able to compute the correct result within the predefined time-frame. The method of Kennel and Jamieson was further adapted, by the authors, to modern processors [30]. The adaptation solves the security issues that arise from the availability of virtualization extensions and multiplicity of execution units.

The authentication protocol is depicted in Figure 6. The initial messages of the protocol carry information about the current configuration of the target machine. Following this exchange, the authentication authority transmits a message containing the challenge code to be executed on the target machine. The target machine executes the challenge, which computes a result that is a cryptographic hash of some memory region, possibly with some additional information. The target machine, concatenates a randomly generated number to the result, encrypts both values with the public key of the authentication authority, and transmits the encrypted message. The authentication authority verifies that the result is correct and was received within a predefined time-frame. If both are true the target machine is considered authentic. The authentication authority then shares some secret information with the target machine. This secret information constitutes a proof of the target's authenticity. The authentication authority encrypts the secret information with a random value obtained from message (3) used as the encryption key, and transmits the encrypted message to the target machine.

5. Encrypted instructions execution

In order to execute an encrypted program, the user must first install the driver, which encapsulates the hypervisor. The driver monitors the PE files (ELF files, in Linux) loaded by the OS, and keeps track of PE files that contain the special encrypted functions section. When the first such PE file is loaded, the driver initializes the hypervisor. During the initialization, the driver communicates with the authentication authority, passes the attestation verification, obtains the cryptographic key, and enters a virtualized state.

The hypervisor is configured to intercept the general protection fault. When a protected program transfers control to an encrypted function, the processor attempts to execute the halt instruction, which induces a general protection fault, thus transferring control to the hypervisor. General protection faults rarely occur during the normal course of program execution, since they usually cause the program to terminate abruptly. Nevertheless, the hypervisor uses the data structures prepared by the encryption tool to test whether the general protection fault occurred during execution of an encrypted function.

The hypervisor injects the interrupt back to the guest, if it was not caused by an encrypted function execution. Otherwise, the hypervisor decrypts the function and starts its execution. Since during its execution, the function is stored in memory in unencrypted form, it is highly important to ensure that no other code has access to the decrypted instructions of the function. We note that in modern processors, several execution units (logical processors) can execute programs concurrently. Therefore, we must ensure that programs executed by all execution units have no access to the unencrypted instructions.

We present two approaches to sensitive functions execution: *in-place execution* and *buffered execution*.

5.1 In-place execution

According to this approach the hypervisor can be in one of two states: ***cold*** or ***hot***. In the cold state the memory does not contain any sensitive information and only the cryptographic key and the hypervisor's state must be protected. This is the regular mode of operation described in section 3. The hypervisor switches to the hot state when the memory contains sensitive information, which cannot be protected by the normal hypervisor memory protection technique (for example, based on EPT), since its physical location is not known (or not constant). EPT (Extended Page Table) is a secondary address

translation facility used by the hypervisor to translate guest physical addresses to actual physical addresses. Switching to hot mode occurs when the hypervisor triggers execution of a decrypted function.

In the following description, we assume that the encryption tool uses ***halt*** as a replacement opcode, but the same is true when the ***software breakpoint*** opcode is used.

At initialization the hypervisor's state is set to cold. In this state, in addition to the regular protection means described in section 3, the hypervisor intercepts general protection faults. An encrypted function, which was overwritten by the NEC consists mainly of halt instructions. Execution of any of these instructions induces a general protection fault, which causes a vm-exit and transfers control to the hypervisor. The hypervisor inspects the source of the general protection fault, and fetches the EC that corresponds to this NEC. Then the hypervisor switches to hot mode and decrypts the EC into its natural location, currently occupied by the NEC (the NEC is saved in a different location for future use).

During the switch to hot mode, the hypervisor freezes all other execution units, and configures itself to intercept all interrupts. This behavior guarantees that the function in its decrypted form cannot be read by any other, potentially malicious, code, simply because no other code can run in hot mode. We note that all the control transfer instructions in the EC are replaced by the halt instruction, which induces a vm-exit.

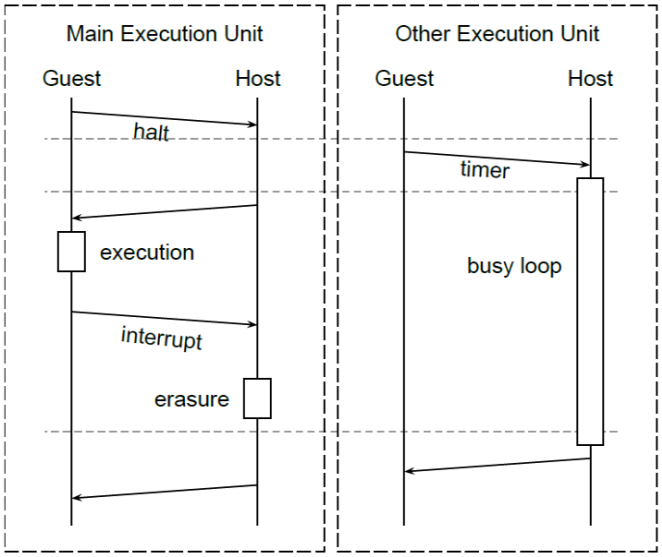

Figure 7. Example of encrypted function execution. The figure depicts two execution units, each with two alternating states: guest and host. The dashed horizontal lines are synchronization barriers, i.e. everything above the line is guaranteed to complete before anything below the line starts.

When a vm-exit occurs in hot mode, the hypervisor first replaces the decrypted function with the NEC, and switches to cold mode. Following this, the hypervisor resumes all the execution units, configures itself to intercept only general protection faults, and returns control to the guest. Figure 7 depicts the control flow during encrypted function execution.

We suggest to freeze other execution units by inducing a vm-exit on each execution unit, and running a busy loop until the hypervisor switches back to cold mode. A vm-exit can be induced either implicitly with a timer or explicitly by sending an inter-processor interrupt (IPI). The former solution is much easier to implement but the later solution is much more efficient.

The hypervisor intercepts interrupts in hot mode by replacing the original interrupt descriptor table (IDT) of the OS with a specially crafted IDT. In this special IDT each handler induces a vm-exit, for example, by executing the CPUID instruction. The hypervisor intercepts this instruction, realizes that an interrupt at vector N occurred and switches to cold mode. The hypervisor proceeds by installing the original IDT and moves the guest's instruction pointer to point to the N^{th} interrupt handler of the original IDT.

5.2 Buffered execution

In the following description, we assume that the encryption tool uses halt as a replacement instruction for NECs and software breakpoint as a replacement instruction for ECs.

According to this approach, the hypervisor has only one state, in which it protects itself as described in section 3. In addition, the hypervisor configures itself to intercept general protection faults. Execution of halt instructions induces a general protection fault, which causes a vm-exit and transfers control to the hypervisor. The hypervisor inspects the source of the general protection fault, and fetches the EC that corresponds to this NEC.

When the EC is resolved, the hypervisor decrypts it into a pre-allocated memory buffer, which is protected by the hypervisor's second-level translation tables (EPT). The decrypted EC will be executed in host mode, thus allowing it to reside in an EPT-protected buffer. Since the decrypted instructions are inaccessible by any other execution unit (in guest mode), there is no need to suspend them. Likewise, since the encrypted instructions are executed inside the hypervisor, there is no need to modify the IDT of the guest. Finally, there is no need to perform the costly transitions to and from the guest after every decryption. All these improve the overall performance of the system by a large factor.

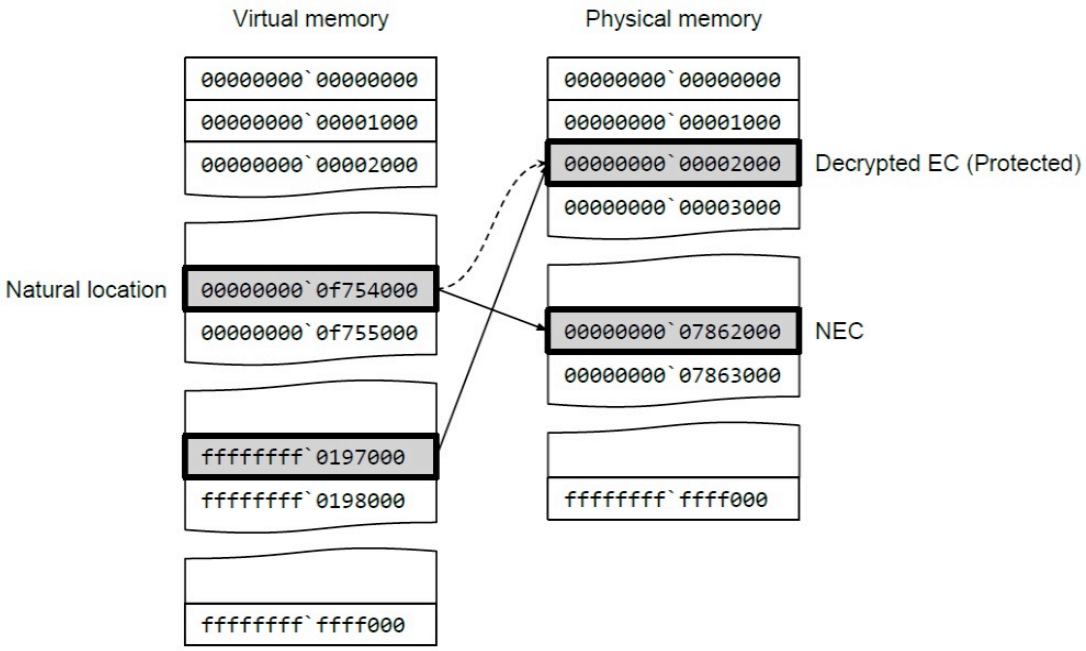

Figure 8. Memory layout during buffered execution. The functions resided at virtual address f754000, which is mapped to the physical address 7862000. The encrypted code is decrypted to virtual address ffffffff`0197000 which is mapped to the physical address 2000. The hypervisor changes the mapping of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

The x86 instruction set architecture defines many memory access instructions as *relative*, meaning that their arguments should not be interpreted as actual memory locations but rather they should be interpreted as offsets from the current value of the instruction pointer. As a consequence, the same instruction may have different interpretations when executed at different locations. Therefore we must execute the decrypted EC at its natural location. In order to achieve this, the hypervisor modifies the virtual page table of the current process by mapping the virtual page containing the NEC to the physical address of the pre-allocated buffer containing the decrypted EC. Figure 8 depicts this transformation.

The control flow during the execution of an encrypted function is illustrated in Figure 9. The process begins when an encrypted function is called. The first instruction in the NEC is the halt instruction; its execution triggers the general protection exception, which induces a vm-exit. The hypervisor prepares the system for buffered execution by performing the following steps: (1) the EC is decrypted into a pre-allocated buffer; (2) the virtual page table is modified to map the natural location of the function to the pre-allocated buffer, as illustrated in Figure 8; (3) the values of the guest registers, which were stored during the vm-exit transition, are restored; (4) the decrypted function is called. The decrypted function

executes until an interrupt occurs. The interrupt can be triggered by a software breakpoint instruction or by some other condition, e.g., a page fault. In both cases the hypervisor suspends the buffered execution by performing the following steps: (1) the values of the registers are stored to a memory region from which they will be restored during vm-entry; (2) the virtual page table is restored to its original state; (3) the decrypted EC is erased. If the interrupt was triggered by a software breakpoint instruction, the hypervisor resumes the guest immediately. However, if the interrupt was triggered by some other condition, the hypervisor injects the interrupt to the guest, and then resumes it. The interrupt injection mechanism allows the hypervisor to delegate the responsibility of interrupt handling to the operating system. Figure 9 illustrates the simple case of software breakpoint interrupt.

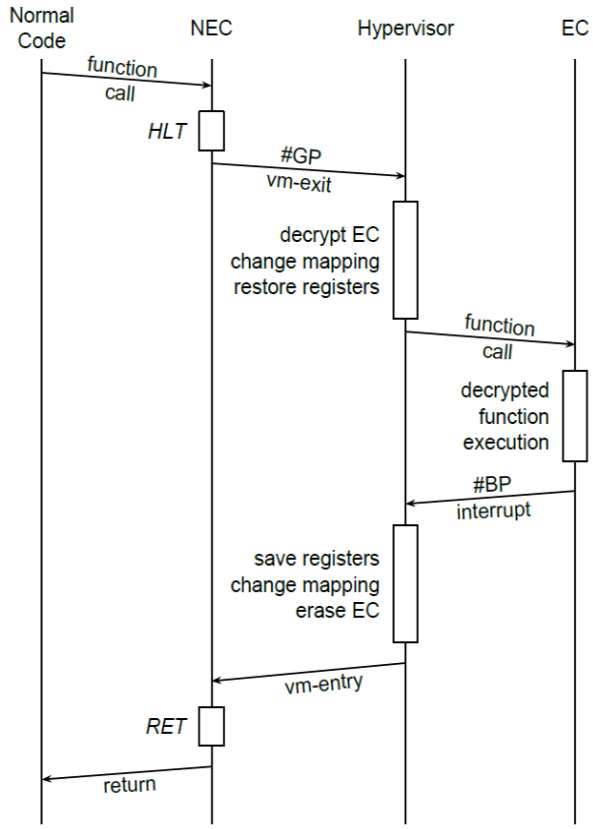

Figure 9. Example of encrypted function execution in buffered execution mode. The figure depicts the control flow during the execution of an encrypted function.

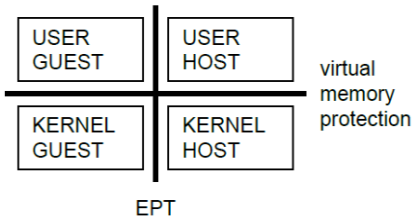

Figure 10. Execution modes. The left column represents the guest mode, while the right column represents the host mode. The lower row represents the kernel mode, while the upper row represents the user mode. The host mode can protect itself from the guest mode through the EPT mechanism. The kernel mode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

This approach is more efficient but potentially less secure than the in-place execution. According to this approach, the decrypted functions are executed inside the hypervisor itself. As a consequence these functions have the same privileges as the hypervisor. In particular, they can read and write memory,

which is otherwise inaccessible to any code external to the hypervisor. One can argue that it is impossible for an adversary to replace the EC with random code, without knowing the cryptographic key. However unfortunately, it is possible that some memory manipulation can be performed indirectly by modifying the data on which the encrypted function works. Nevertheless, although possible, it seems to be extremely difficult to manipulate the behavior of unknown code through its data. Possible solutions to this problem will be discussed in our future research.

6. Performance

This section presents a performance analysis of the two execution methods that were described in section 5.

We first measured the direct overhead associated with executing an encrypted function. To do that we created a function `f()` of size 128 bytes. The function's first instruction is a return instruction, therefore, once activated, the function immediately returns to the caller. In the executable file we encrypt `f()` and measure the number of CPU cycles used in a call to `f()`. Since `f()` is encrypted, calling `f()` entails a transfer from "cold" mode to "hot" mode, i.e. VM_EXIT to the hypervisor, decryption of `f()`'s contents execution of `f()` (in this case basically zero cycles since the first instruction is an immediate return) and then restoring to "cold" mode. Measurements of this full-cycle were averaged over 10000 trials with an average of 7100 cycles when using "buffered" mode and 23,000 cycles when using "in-place" mode.

To measure the overhead associated with real-world applications, we decided to use standard benchmarks as the model. The measurements were performed by encrypting several of the major functions in standard benchmark programs and comparing the performance results of each benchmark when executed with and without those functions encrypted. Two performance measurements were obtained for benchmarks that were run with an encrypted function: (a) using "In-Place Execution" and (b) using "Buffered-Execution".

System overhead, as a result of running encrypted code over the hypervisor, is attributed to actions that need to take place in the hypervisor during a VM_EXIT. This occurs when (a) an encrypted function is called; (b) a call is made from within an encrypted function to a non-encrypted function; a return occurs from the calls in (a) or (b). In (a) the function needs to be decrypted and the processor is put into "hot" mode: when the "In-Place" method is used other processors need to be frozen; when "buffered" mode is used the hypervisor needs to remap the execution pages. In (b) and (c) the operation is reversed by clearing decrypted-memory and putting the processor back into "cold" mode. Therefore, overhead is closely related to the number of transitions into and out of "hot" mode.

Additional overhead can be observed as a result of activating the hypervisor without regard to activities required to support executing encrypted software. This overhead is attributed to the fact that the system is running over a hypervisor, which activates *secondary level address translation* (SLAT) that implies overhead as a result of the additional translation required for memory access, as well as needing to intercept some mandatory events.

Performance measurements of encrypted software execution overhead were conducted by running well-known benchmarks on a multiprocessor system with and without encrypted functions.

We chose the "Phoronix Test Suite" [31] as our benchmark suite. A variety of test benchmarks were selected to reflect different types of loads, such as: CPU intensive, graphics, disk-access and network activities. The tests were performed on a system with the following configuration:

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

We have performed three tests. In each test, we have selected an application and encrypted several central functions. Table 1 summarizes the information about the encrypted function in each application.

The first application, "Parallel BZIP2 Compression", is CPU intensive. It measures the time needed to compress a file (a .tar package of the Linux kernel source code) using BZIP2

compression. The second application, "Unpacking the Linux Kernel", measures how long it takes to extract the .tar.bz2 Linux kernel package. The third application is "X11 – 500px PutImage Square". The package "x11perf" is a very basic performance/regression test for X.Org (Window System).

Each of the benchmark tests was executed after a full system reboot (to ensure a "clean" system) and measured under the following conditions: (a) non-encrypted executable without a hypervisor active; (b) non-encrypted executable with a commercial hypervisor (VMWare) active; (c) non-encrypted executable with TrulyProtect thin-hypervisor active; (d) Encrypted executable using "In-Place" mode; and (e) Encrypted executable using "Buffered" mode. Each activation of a "Phoronix Test Suite" benchmark generates multiple runs of the benchmark to gather significant statistics.

Table 2 presents the results that were measured during benchmark execution in various configurations. The two leftmost columns describe the configuration in which the test was executed. The third column specifies the parameter that was measured. The three rightmost columns contain the values that were measured for each parameter. The table is divided into five parts: (a) No hypervisor – where measurements were performed on a non-encrypted executable without an active hypervisor; (b) vmWare HV active and KVM HV active – where measurements were performed on a non-encrypted executable with a commercial hypervisor (vmWare and KVM); (c) TP HV Active – where measurement were performed with TrulyProtect thin-hypervisor; (d) Overhead Calculation – this part summarizes the first three parts; (e) Net overhead calculations – this part presents the overhead of the in-place and the buffer decryption methods after subtraction of the overhead associated with TrulyProtect hypervisor.

Application	Function name	Size (in bytes)
Parallel BZIP2 Compression	BZ2_bzBuffToBuffCompress	317
	BZ2_bzCompressInit	588
	BZ2_bzCompress	380
	BZ2_bzCompressEnd	123
Unpacking the Linux Kernel	extr_init	94
	run_decompress_program	443
	tar_checksum	175
	extract_finish	47
	checkpoint_finish	63
X11 500px PutImage Square	InitPutImage	93
	InitGetImage	140
	DoPutImage	350

Table 1. Encrypted functions summary.

The third part is further subdivided into three parts: (i) Non protected – where a non-encrypted executable was measured; (ii) In-Place – where an encrypted executable was executed using the in-place decryption method; (iii) Buffered – where an encrypted executable was executed using the buffered decryption method.

The fourth part compares the execution times of a non-encrypted executable to four other modes of execution: (i) a non-encrypted executable while a commercial hypervisor is active; (ii) a non-encrypted executable while TrulyProtect thin-hypervisor is active; (iii) an encrypted executable which is executed using the in-place decryption method; (iv) an encrypted executable which is executed using the buffered decryption method. A graphical representation of this data appears in figures 11. Figure 12 presents the overhead of the in-place and the buffer decryption methods after subtraction of the overhead associated with TrulyProtect hypervisor.

Overhead was calculated by solving for the degradation in percent relative to the reference benchmark result as measured without the hypervisor activated.

			Parallel BZIP2 Compression	Unpacking the Linux Kernel	X11 500px PutImage Square
No HV	Not Protected	Execution	26.58 secs	10.31 secs	2822 ops/sec
vmWare HV Active	Not Protected	Execution	28.92 secs	14.83 secs	1643 ops/sec
KVM HV Active	Not Protected	Execution	28.39 secs	11.4 secs	905 ops/sec
TP HV Active	Not Protected	Execution	26.92 secs	11.81 secs	2795 ops/sec
	In-Place	Execution	31.74 secs	16.6 secs	1997 ops/sec
		VM_EXITs	222	129663	170857
		Decryptions	64	64743	85263
	Buffered	Execution	27.07 secs	12.05 secs	2667 ops/sec
		VM_EXITs	174	64743	107316
		Decryptions	64	64743	107316
Overhead Calculations	vmWare HV		9%	44%	42%
	TP HV		1%	15%	1%
	In-Place		19%	61%	29%
	Buffered		2%	17%	5%
Net Overhead	In-Place		18%	46%	28%
	Buffered		1%	2%	5%

Table 2. Test results.

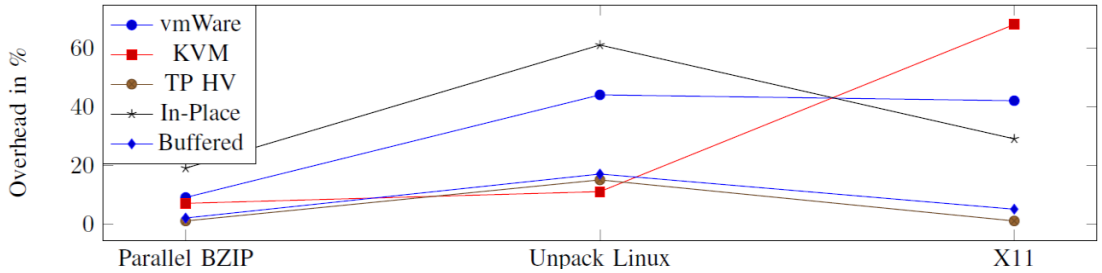

Figure 11. Overhead calculation relative to no-hypervisor benchmarks.

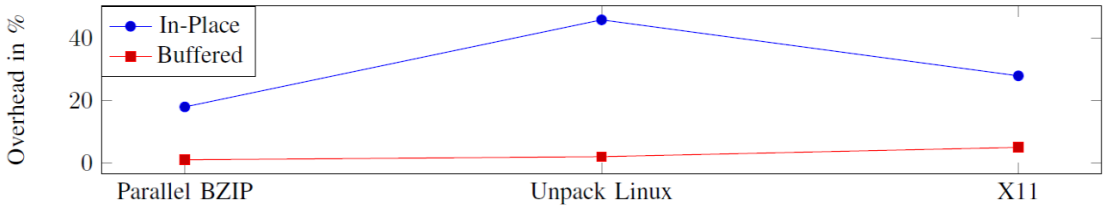

Figure 12. Net encrypted execution overhead.

7. Future work

As was explained above, the buffered execution method is superior to the in-place execution method in terms of performance. Unfortunately, the buffered execution method allows an adversary to access regions of memory that are normally protected by the hypervisor. Consider the *memcpy* function, for example. Assume that this function is encrypted and is now being executed by the hypervisor in buffered execution mode. By specifying the address of the VMCS structure in the *source* or *destination* argument, an adversary can inspect and modify the control structures of the hypervisor. Moreover, since the

hypervisor executes in kernel mode, the protected function can access OS memory region and execute privileged instructions.

Fortunately, the x86 instruction set architecture provides a great variety of memory protection mechanisms, which can be utilized by the buffered execution method. One such mechanism is the virtual memory protection, which is available in both 32- and 64-bit execution modes. The virtual memory protected mechanism allows to specify a separate set of accessibility rights for kernel mode and user mode. Similarly, the hypervisor's memory protection (virtualization, to be precise) mechanism, called the Extended Page Table (EPT) on Intel processors, allows to specify a separate set of accessibility rights for host mode and guest mode. The different modes of execution and the protection mechanisms are summarized in Figure 10.

The in-place execution method utilizes the EPT to protect hypervisor's control structures and other sensitive data from an adversary. We propose to use the virtual memory protection mechanism in the buffered execution method. In particular, the buffered execution method can execute the decrypted function in user mode inside the host mode (the upper right block in Figure 10); this mode is not used by the system described in this paper. In this mode we can prevent attempts to execute privileged instructions or access the hypervisor's control structures.

The hypervisor can transit to this mode by executing the iret instruction, which is usually used to terminate an interrupt handler. This instruction modifies the execution location and the execution mode (from kernel to user). Since the execution takes place in host mode, interrupts cannot be intercepted by the hypervisor through configuration of the VMCS. The hypervisor is forced to use the IDT, which allows the kernel to specify the interrupt service routines for each of the 256 interrupt vectors. Upon interrupt, the interrupt service routine can decide whether to handle the interrupt inside the hypervisor or inject it to the guest.

We believe that the described approach will substantially improve the security of the buffered execution method, thus making it absolutely superior to in-place execution.

8. Conclusions

We present research pertaining to the methodologies of executing encrypted native machine-code, where decryption and execution are done on the fly and secure with a thin hypervisor. Two alternative methods are considered: *in-place* and *buffered* – that trade security for performance. The in-pace method executes decrypted-code in guest mode, thereby limiting the functionality of the decrypted function to whatever a guest may perform. In buffered execution method, the decrypted function executes in host mode, potentially incurring the risk of a rogue implementation accessing sensitive memory areas. For this reason the in-place method is considered safer. However, in modern multi-processor systems, the in-place method requires controlling (freezing) other execution units, while a single execution unit executes decrypted code. This requires larger overhead when compared to the buffered method and thus has a performance toll. Larger overhead is expected to be more significant for larger functions. The reason for this is related to the fact that overhead is acquired during transitions between cold to hot and hot to cold modes in the in-place method, as compared to transitions between host-execution of decrypted code and guest-execution of interrupts. Larger functions acquire more transitions, therefore overhead is more prominent in the in-place method. Given these results our conclusions are to use the (safer) in-place methodology for short functions (smaller than 1000 bytes). For larger functions (larger than 1000 bytes), allow a user-defined switch in the encryption tool to prefer security, in which case in-place shall be used, or performance, in which case buffered shall be used. In future work we plan to augment the buffered method to overcome its potential security flaws and render it the single and best alternative to use.

9. References

[1] Themida, http://www.oreans.com/, Oreans.
[2] VMProtect, http://vmpsoft.com/, VMProtect Software.
[3] R. Rolles, “Unpacking Virtualization Obfuscators,” in Proceedings of the 3rd USENIX Conference on Offensive Technologies, ser. WOOT’09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1.
[4] L. Bohne, “Pandora’s Bochs: Automated Unpacking of Malware,” 2008.

[5] D. Schellekens, B. Wyseur, and B. Preneel, "Remote Attestation on Legacy Operating Systems with Trusted Platform Modules," Sci. Comput. Program., vol. 74, no. 1-2, pp. 13–22, Dec. 2008.
[6] S. Pearson, Trusted Computing Platforms: TCPA Technology in Context. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.
[7] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A Trusted Open Platform," Computer, vol. 36, no. 7, pp. 55–62, Jul. 2003.
[8] C. Tarnovsky, "Semiconductor Security Awareness Today and yesterday," in Blackhat, 2010. [Online]. Available: https://www.youtube.com/watch?v=WXX00tRKOlw
[9] C. Tarnovsky, "Attacking TPM part two," in Defcon, 2012. [Online]. Available: https://www.youtube.com/watch?v=Ed 9p7E4jIE
[10] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, "Truly-Protect: An Efficient VM-Based Software Protection," Systems Journal, IEEE, vol. 7, no. 3, pp. 455–466, 2013.
[11] M. Kiperberg and N. J. Zaidenberg, "Efficient Remote Authentication," in The Journal of Information Warfare, vol. 12, no. 3, 2013.
[12] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual, 2007, vol. 3.
[13] "AMD64 Architecture Programmer's Manual Volume 2: System Programming," AMD, 2010.
[14] M. Pietrek, "An in-depth look into the Win32 portable executable file format," in MSDN Mag. 17, 2, 2002, pp. 80–90.
[15] E. Youngdale, "Kernel korner: The elf object file format by dissection," Linux Journal, vol. 1995, no. 13es, p. 15, 1995.
[16] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," Commun. ACM, vol. 17, no. 7, pp. 412–421, Jul. 1974.
[17] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "Bitvisor: A thin hypervisor for enforcing i/o device security," in Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 121–130.
[18] Y. Chubachi, T. Shinagawa, and K. Kato, "Hypervisor-based Prevention of Persistent Rootkits," in Proceedings of the 2010 ACM Symposium on Applied Computing, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 214–220.
[19] R. Kennell and L. H. Jamieson, "Establishing the Genuinity of Remote Computer Systems," in Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 21–21.
[20] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 1–16.
[21] Q. Yan, J. Han, Y. Li, R. H. Deng, and T. Li, "A software-based root-of-trust primitive on multicore platforms," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 334–343.
[22] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: softWare-based attestation for embedded devices," in Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on, May 2004, pp. 272–282.
[23] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the Difficulty of Software-based Attestation of Embedded Devices," in Proceedings of the 16th ACM Conference on Computer and Communications Security, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 400–409.
[24] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in Proceedings of the 5th ACM Workshop on Wireless Security, ser. WiSe '06. New York, NY, USA: ACM, 2006, pp. 85–94.
[25] Y. Yang, X. Wang, S. Zhu, and G. Cao, "Distributed software-based attestation for node compromise detection in sensor networks," in Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems, ser. SRDS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 219–230.
[26] D. Ionescu, "Microsoft bans up to one million users from xbox live," PC World, Tech. Rep., 2009. [Online]. Available: http://www.pcworld.com/article/182010/xbox users banned.html

[27] Sony, “Information on banned accounts and consoles,” Sony consumer electronics, Tech. Rep., accessed on may 2015. [Online]. Available: https://support.us.playstation.com/app/answers/detail/aid/ 1260/~/information-on-banned-accounts-and-consoles
[28] Brian, “Nintendo starting to ban pirates from online services on 3ds,” Nintendo everything, Tech. Rep., 2015. [Online]. Available: http://nintendoeverything.com/ nintendo-starting-to-ban-pirates-from-online-services-on-3ds
[29] Wikipedia, “An analysis of proposed attacks against genuinity tests,” Tech. Rep., accessed on May 2015. [Online]. Available: http://en.wikipedia.org/wiki/Warden (software)
[30] M. Kiperberg, A. Resh, and N. J. Zaidenberg, “Remote Attestation of Software and Execution-Environment in Modern Machines,” in CSCloud, 2015.
[31] M. Larabel and M. Tippett, “Phoronix test suite,” Phoronix Media, Tech. Rep., accessed on June 2, 2016. [Online]. Available: http://www.phoronix-test-suite.com/
[32] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, C. Rozas, "Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave," Proceedings of the Hardware and Architectural Support for Security and Privacy, Seoul, Republic of Korea: ACM, 2016, pp. 1-9
[33] M Kiperberg, N. J. Zaidenberg “Efficient Remote authentication” Journal of information warfare October 2013
[34] A. Resh, N. J. Zaidenberg “Can keys be hidden inside the CPU on modern windows host” ECIW 2013 pages 231-235