JYVÄSKYLÄ STUDIES IN COMPUTING
228

Michael Kiperberg

Preventing Reverse Engineering of Native and Managed Programs

JYVÄSKYLÄN YLIOPISTO

JYVÄSKYLÄ STUDIES IN COMPUTING 228

Michael Kiperberg

Preventing Reverse Engineering of Native and Managed Programs

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi yliopiston Agora-rakennuksen auditoriossa 1
joulukuun 15. päivänä 2015 kello 10.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
in building Agora, auditorium 1, on December 15, 2015 at 10 o'clock.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2015

Preventing Reverse Engineering of Native and Managed Programs

JYVÄSKYLÄ STUDIES IN COMPUTING 228

Michael Kiperberg

Preventing Reverse Engineering of Native and Managed Programs

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2015

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

URN:ISBN:978-951-39-6437-5
ISBN 978-951-39-6437-5 (PDF)

ISBN 978-951-39-6436-8 (nid.)
ISSN 1456-5390

Jyväskylä University Printing House, Jyväskylä 2015

ABSTRACT

Kiperberg, Michael
Preventing Reverse Engineering of Native and Managed Programs
Jyväskylä: University of Jyväskylä, 2015, 60 p.(+included articles)
(Jyväskylä Studies in Computing
ISSN 1456-5390; 228)
ISBN 978-951-39-6436-8 (nid.)
ISBN 978-951-39-6437-5 (PDF)
Finnish summary
Diss.

One of the important aspects of protecting software from attack, theft of algorithms, or illegal software use is eliminating the possibility of performing reverse engineering. One common method used to deal with these issues is code obfuscation. However, it is proven to be ineffective. Code encryption is a much more effective means of defying reverse engineering, but it requires managing a cryptographic key available to none but the permissible users. The thesis presents a system for managing cryptographic keys in a protected environment and supporting execution of encrypted code. The system has strong security guarantees. In particular, the cryptographic keys are never stored on the target machine, but rather delivered to it from a remote server, upon a successful verification of its authenticity. The keys and the decrypted instructions are protected by a thin hypervisor at all times. The system allows the encryption and execution of both native and Java code.

During native code execution, the decrypted instructions are inaccessible to a potentially malicious code. This is achieved by either preventing execution of any other code or by protecting the memory region containing the decrypted instructions during their execution.

Java programs, unlike native programs, are not executed directly by the processor, but are interpreted (and sometimes compiled) by the Java Virtual Machine (JVM). Therefore, the JVM will require the cryptographic key to decrypt the encrypted portions of Java code, and there is no feasible way of securing the key inside the JVM. The thesis proposes to implement a Java bytecode interpreter inside the secure environment, governed by a thin hypervisor. This interpreter will run in parallel to the standard JVM, both cooperating to execute encrypted Java programs.

Keywords: trusted computing, virtualization, hypervisor, thin hypervisor, Java, remote attestation, interpretation, buffered execution

Author

Michael Kiperberg
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Supervisors

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

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

Reviewers

Professor Vincenzo Piuri
Department of Computer Science
The University of Milan
Italy

Professor Samuel Itzikowitz
The School of Computer Science
The College of Management
Israel

Opponent

Doctor Christian Grothoff
Decentralise team
Inria
France

ACKNOWLEDGEMENTS

First of all, I would like to thank my scientific supervisors, Prof. Pekka Neittaanmäki and Dr. Nezer Zaidenberg, for their guidance, help and moral support throughout the work on this thesis.

I am indebted to the external reviewers of my thesis for their valuable comments and suggestions. I would also like to thank Amit Resh, Asaf Algawi and Roee Leon for co-authoring the joint publications included in this thesis.

I greatly appreciate the support of the COMAS Graduate School, which provided funding for this research. I am also grateful to the Department of Mathematical Information Technology, which financially supported numerous conference trips.

I am thankful to my friends for their moral support and interest in my work.

Finally, I wish to thank my parents, Lubov and Yakov, and my brother Arthur, without whom this work would not have been possible, for their love and encouragement.

Jyväskylä
November 30, 2015
Michael Kiperberg

LIST OF FIGURES

LIST OF TABLES

CONTENTS

LIST OF INCLUDED ARTICLES

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

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

PIII Kiperberg, M.; Zaidenberg, N.J.. Efficient Remote Authentication. *The Journal of Information Warfare , vol.12, no.3*, 2013.

PIV Averbuch, A.; Kiperberg, M.; Zaidenberg, N.J.. Truly-Protect: An Efficient VM-Based Software Protection. *Systems Journal, IEEE , vol.7, no.3, pp. 455-466*, 2013.

PV Averbuch, A.; Kiperberg, M.; Zaidenberg, N.J.. An efficient VM-based software protection. *Network and System Security (NSS), 2011 5th International Conference, pp. 121-128*, 2011.

PVI Kiperberg, M.; Resh, A.; Algawi, A.; Zaidenberg, N.J.. System for Executing Encrypted Java Programs. *IEEE Transactions on Dependable and Secure Computing*, Submitted.

PVII Kiperberg, M.; Leon, R.; Resh, A.; Zaidenberg, N.J.. System for Executing Encrypted Native Programs. *IEEE Symposium on Security and Privacy*, Submitted.

The idea of encrypted code execution, which is described in articles [PV] and [PIV], was devised together with Dr. Nezer Zaidenberg. The author designed and implemented the encryption tool and the decryption system, and described their design and performance analysis in articles [PV] and [PIV].

Article [PIII], which was written mainly by the author, discusses the security problems of the system that was presented in articles [PV] and [PIV], and presents possible solutions to those problems. The data presented in the article was collected by the author.

The author devised a hypervisor detection method, which can be embedded into the remote authentication scheme described by Kennell and Jamieson [KJ03]. In addition, the author adapted the remote authentication scheme to multi-core processors. The improvements to the original scheme are described in article [PI], which was written mainly by the author. The co-authors invented and presented the performance counters chaining method.

The author implemented the hypervisor-based decryption system and the encryption tool, as described in article [PVII], which was written mainly by the author. The co-authors implemented the AES decryption scheme and the buffered execution method.

The author devised and designed a system for executing encrypted Java programs. The author implemented the decryption system: both the hypervisor module and the user-mode module. The encryption tool was designed by the author and implemented by the article's co-authors. Article [PVI], which was mainly written by the author, describes the system. The co-authors described the encryption tool.

1 INTRODUCTION

This chapter describes the research area of the thesis, introduces the main concepts, and outlines the structure of the thesis.

One of the important aspects of protecting software from attack, theft of algorithms, or illegal software use is eliminating the possibility of performing reverse engineering. One common method used to deal with these issues is code obfuscation. However, it is proven to be ineffective. Code encryption is a much more effective means of defying reverse engineering, but it requires managing a cryptographic key available to none but the permissible users. The thesis presents a system for managing cryptographic keys in a protected environment and supporting execution of encrypted code. The system has strong security guarantees. In particular, the cryptographic keys are never stored on the target machine, but rather delivered to it from a remote server, upon a successful verification of its authenticity. The keys and the decrypted instructions are protected by a thin hypervisor at all times. The system allows the encryption and execution of both native and Java code.

1.1 Obfuscation

Software obfuscation is a set of techniques used to transform one program into another, such that both have the same semantics but the latter is much harder for a human reader to comprehend. An overview of different obfuscation techniques and other digital rights managements approaches can be found in [PII]. The vendors of obfuscation technologies rarely publish the design of their products, thus preventing their public scrutinization. Nevertheless, some researchers deduce the design of the obfuscating transformation from the resultant program and report their findings in academic press. Usually their reports include not only a detailed description of the transformation but also an outline of the design flaws. Sometimes, the reporters devise an automatic or a semi-automatic tool which is able to reverse the obfuscating transformation.

1.1.1 Instruction Set Architecture

An instruction set architecture (ISA) of a processor is a definition of the capabilities of the processor and the means by which those capabilities can be utilized by a program. In particular, the ISA defines the format and the meaning of each instruction, specifies the arguments of the instruction and the result of its execution.

An ISA can be implemented in hardware, like x86, or in software, like Java bytecode. An execution of a program written for one ISA on a hardware processor of a different ISA is called emulation. Emulation is inherently inferior to direct execution in terms of performance. Nevertheless, it is still widely used in situations where the benefits of emulation (portability, for instance) outweigh its performance penalty. Emulation was extensively studied and two main approaches evolved: interpretation and binary translation. In practice, emulators usually include both an interpreter and a binary translator, and apply each of them where appropriate.

1.1.2 Obfuscation Methods

The most general form of obfuscation, and the one which is generally used in practice, is obfuscation through emulation. The obfuscating transformation translates the original program, in its binary form, to another ISA. The transformation then bundles the transformed code with an emulator of the new ISA. This bundle constitutes the new program, which is then delivered to consumers.

Any ISA allows the performance of operations on immediate values (constants), which are usually supplied as instruction arguments. Obfuscating these values is highly important since they may guide an adversary through the program and lead to its (at least partial) understanding. Many obfuscating transformations translate instructions with immediate values not only by retargeting them at a different ISA, but also by replacing the immediate value with a sequence of instructions whose computation results in this value.

Devising a new ISA for each obfuscating transformation is not practical. Therefore, the obfuscating tools provide only a small set of target ISAs. In addition to the target ISA, the user of such tools can choose a permutation on the instruction encoding, thus making the byte 0x12 mean "addition" according to one permutation, and "multiplication" according to another.

1.1.3 Breaking Obfuscation

Obfuscation based on ISA retargeting is probably the most challenging to break. In fact, even today many malicious programs use this type of obfuscation to prevent their detection by anti-virus and anti-malware software.

Rolles has proposed [Rol09] a method to reconstruct the original program from an obfuscated one. The method has three stages. At the first stage a human adversary studies the obfuscated program, locates the interpreter, and analyzes

it.

The interpreter consists of two main parts: a dispatch loop and instruction handlers. The dispatch loop performs the following three operations until the program terminates: (1) fetch the next instruction, (2) decode it, and (3) dispatch to the correct handler. The instruction handlers modify the internal state of the interpreter according to the required operation. For example, the *add* instruction handler may add the value of one variable to another variable (the variables may represent registers of the emulated ISA). Some optimization may affect this conceptual structure of the interpreter.

At the second stage the human adversary captures the meaning of each instruction handler in a few original ISA instructions. The result of this stage is a dictionary from a previously unknown ISA to the original, known, ISA. This is the last step performed by a human adversary.

The third stage is performed automatically. An automatic tool uses the information gathered by the human adversary to find, in the transformed program, the dispatch loop and the instruction handlers. Then the automatic tool constructs a dictionary from the instruction encodings to the instruction handlers. Together with the dictionary that maps instruction handlers to sequences of original ISA instructions, this information is sufficient to translate the program to the original ISA.

Researchers claim that with additional sophistication of the method described above it is possible to recreate a program whose structure resembles the structure of the original program. We note that human intervention is required only when a new ISA is devised, which requires a human intervention as well, so roughly the same effort is required from the attacking and defending sides.

1.2 Applicability of Obfuscation

Recreation of obfuscated programs requires a modest amount of work. Therefore, obfuscation cannot be used to protect programs of high value, such as military systems (high security value), algo-trading program (high financial value), or even programs of moderately high (financial) value, such as video games.

1.2.1 Digital Rights Protection

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 the program, which leaves them vulnerable to misuse.

One way of preventing circumvention of the software licensing program, may be using a method of obfuscation. The term obfuscation refers to making

software instructions difficult for humans to understand by deliberately cluttering the code with useless, confusing pieces of additional software syntax or instructions. However, even when changing the software code and making it obfuscated, the content is still readable to the skilled hacker.

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 [SWP08, Pea02, ELM+03], but may have significant deficiencies, mainly due to the 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 [Tar10, Tar12].

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

1.2.2 Military

Software produced by military and intelligence organizations may contain sensitive or classified information and algorithms. In this case the potential adversary is well equipped, and is not limited by time or other resources, since it is supported by the government of a hostile state.

Our experience suggests that, generally, the only effective countermeasure is the physical isolation of the software and its environment from a potential adversary. In case of exposure, the software and its environment are physically destroyed, either remotely or by the operator of this software. There are two main disadvantages to this approach. The first is the danger of unintentional activation of the destruction mechanism. The second is the failure to destroy the software when an exposure is detected, either due to the inability of the human operator to trigger the destruction sequence, or failure of the (possibly damaged) destruction mechanism itself. Therefore, a better solution is required — a solution which can guarantee the secrecy of the software information by design, and which does not require any special or dangerous actions.

1.3 Encryption

Traditionally, when a sensitive information has to be placed in a potentially hostile environment, it is first encrypted with a key, which is kept in a trusted environment at all times. Inspection and manipulation of the information requires the key. Previously, researchers suggested [Bes80] a processor architecture that includes a cryptographic module, which can be used to decrypt the instructions as

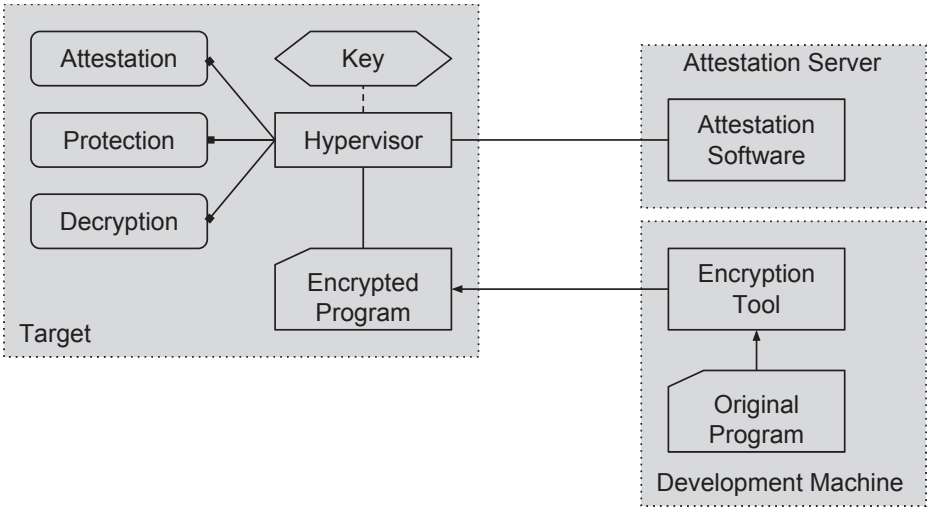

FIGURE 1 Native code protection system. The original program is encrypted before its distribution. The encryption key is stored in the attestation server, which delivers it to the hypervisor in the target machine upon successful attestation. The hypervisor is responsible for decrypting the protected code and protecting itself, the decryption key and the decrypted code from an adversary.

the first step in their execution process. The cryptographic keys are embedded in the processor during the manufacturing process and are known only to the manufacturer. This suggestion is partially realized in modern processors. For example, Intel announced a new extension to their x86 processors, the software guard extension (SGX), which adds cryptographic modules and embeds cryptographic keys that protect a region of memory inside a program from unauthorized access. The protected regions of memory are stored in encrypted form. This extension, by its design, is primarily targeted at data protection rather than code protection, although it is probably possible to use this technology for code protection as well. Unfortunately, SGX is currently unavailable to computer manufacturers and its future availability on the widespread processor families is uncertain.

1.4 Our Method

1.4.1 Native Code

This work provides a detailed description of a native code protection method which is based on encryption and can be implemented on commonplace processors. Our method consists of several components, that together provide protection for both the decryption key and the encrypted program. Figure 1 outlines the relationships between the different components of the system. Conceptu-

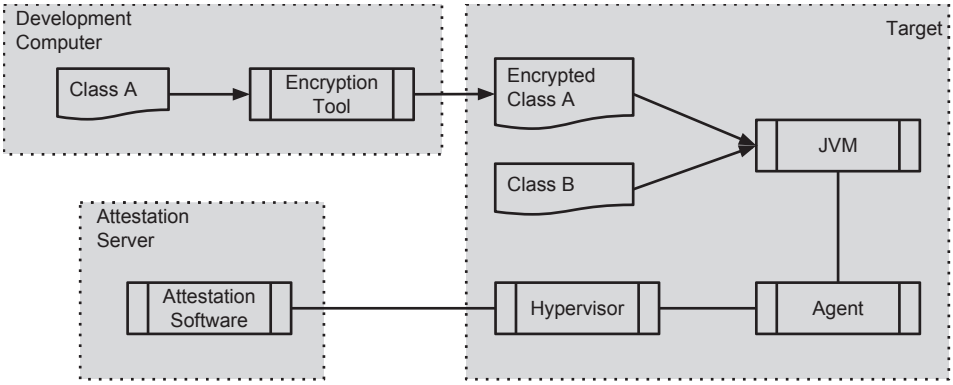

FIGURE 2 Java code protection system. The encryption tool transforms regular Java classes into encrypted ones. Regular and encrypted Java classes are then loaded by the JVM. The JVM loads a JVM TI agent through the JVM TI interface. The agent links the hypervisor to the JVM and assists in the interpretation process. The agent communicates with the JVM through JVM TI and JNI. The communication between the agent and the hypervisor is based on hypercalls and execution frames. The hypervisor receives the decryption key from a remote server, which attests the validity of the hypervisor and the hardware on which it executes.

ally, the components are deployed on three computers. The program is originally compiled on the development machine and encrypted using the encryption tool. Then the program is distributed to consumers, where it gets installed on the target computer. The decryption key is stored in the attestation server, which delivers it to the target computer upon its successful authentication. This scheme was originally described in [PV, PIV].

In addition to the encrypted program, the installation package installs a hypervisor, which has three aspects: attestation, protection, and decryption. The attestation aspect refers to the hypervisor's ability to participate in the remote attestation protocol. The purpose of this protocol is to verify the authenticity of the target hardware and software, and deliver the cryptographic keys from the attestation server to the target computer. The protection and decryption aspects refer to the hypervisor's ability to decrypt programs and execute them while guaranteeing the secrecy of the decrypted instructions and the cryptographic key. The next chapters describe the design and implementation details of the three aspects.

1.4.2 Managed Code

In recent years, programs that are targeted at managed execution environments have become widespread [DKGC07]. Unlike regular (native) programs, managed programs cannot be executed directly by the CPU and, therefore, require a special (native) program to interpret the managed program. Managed execution environments are superior to native environments in memory management, debug-

ging and profiling support. For these reasons, managed execution environments have become popular among developers of desktop and mobile applications.

While it is possible to guarantee that a sequence of native instructions cannot be intercepted (read or modified) during its execution by a CPU, such a guarantee cannot be made for a managed execution environment, since an unexpected behavior can be introduced into the software that implements the managed execution environment. This work describes a technique for executing safely encrypted managed programs on the available managed execution environments.

Executing an encrypted managed program is more challenging and this results in a more complex system design, which is captured by Figure 2. Similarly to the design of the native system, this design consists of three conceptual computers: an encryption tool, a hypervisor, and an attestation software. The hypervisor still has the three aspects described above. This system extends the native system by introducing two new components: the JVM, which is the standard execution environment for Java programs, and Agent, which is a JVM TI agent capable of inspecting and modifying the internal state of the program and the JVM.

In order to support near-optimal performance of executing the unencrypted parts of a Java program, the proposed system executes them on a regular JVM. The JVM TI agent intercepts attempts to execute encrypted code and transfers control to the hypervisor. The hypervisor decrypts and executes the code until it reaches an instruction, whose execution requires cooperation with the JVM. At this point, the hypervisor returns control to the JVM TI agent and passes it the decrypted instruction. The JVM TI agent executes the instruction and transfers control back to the hypervisor. In contrast to the native code execution, this design leaks some secret information. This work provides estimates on the amount of leaked information and discusses techniques that can reduce this amount.

1.5 Structure

The structure of the thesis is as follows: the next chapter overviews virtualization in general and its applicability to program protection. Chapter 3 presents a remote authentication method capable of authenticating software and hardware of modern machines. Chapters 4 and 5 describe virtualization-based methods for native and Java programs' protection. An overview of the articles included in the thesis is provided in chapter 7. Finally, chapter 6 summarizes the contribution and limitations of the thesis and outlines the directions of further research. Figure 3 depicts the relationships between the chapters and the describe system.

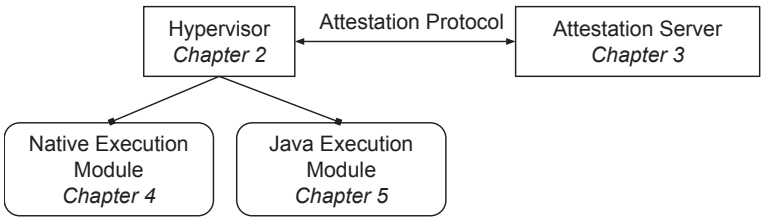

FIGURE 3 Relationships between the chapters and the described system. The attestation server authenticates the hypervisor which embeds either the native execution module or the Java execution module.

2 HYPERVISORS

A hypervisor, also referred to as a Virtual Machine Monitor (VMM), is software, which may be hardware assisted, that manages multiple virtual machines on a single system [PG74]. 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*.

2.1 Hardware Assisted Virtualization

Hypervisors have been in use as early as the 1960s on IBM mainframe computers [Cre81]. After 2005, Intel and AMD introduced hardware support for virtualization (Intel VT-X [Int07], AMD AMD-V [AMD10]), which allowed the implementing of hypervisors in the ubiquitous PC platforms. There are slight differences between Intel's and AMD's implementation of the x86 virtualization extension. In this work we will discuss only Intel's implementation and mention the differences where they are important for the discussion.

In order to support multiple OS guests, a hypervisor must unobtrusively intercept OS access to hardware resources so it can attend to them itself. The hypervisor can then manage hardware allocations that maintain proper separation between the guests. The guest OS is unaware of the hypervisor's intervention, as it experiences a normal hardware access cycle. The only distinction being the elapsed time, since the hypervisor mediation has a time-toll. This property led to a debate regarding the detectability of a hypervisor [BYDD$^+$10, RT07, RT08, Fer07].

A virtual machine control structure (VMCS) is defined for each virtual environment managed by a virtual machine monitor (VMM). This structure defines the values of privileged registers, the location of the interrupt descriptors table,

and additional values that constitute the internal state of the virtual environment. In addition, this structure defines the events that the VMM is configured to intercept, and the address of the function that should handle the interception. The act of control transfer from the virtual environment to a predefined function is called vm-exit and the act of control transfer from the function back to the virtual environment is called vm-entry. Upon vm-exit the function can determine the reason of the vm-exit by examining the fields of the VMCS and altering them, thus altering the state of the virtual environment as it wishes. Finally, the VMCS can define a mapping between the physical memory as it is perceived by the virtual environment and the actual physical memory. As a consequence, the VMM can prevent access to some physical pages by the virtual environment. Moreover, the virtual environment will be unaware of this situation.

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

2.2 Thin Hypervisor

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 [SET+09, CSK10]. 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 4 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.

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 or the cryptographic key.

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.

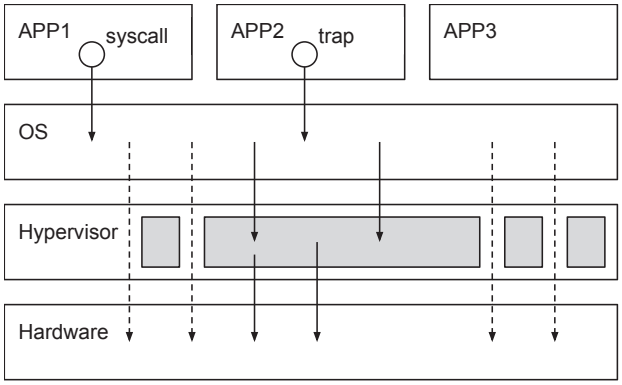

FIGURE 4 Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions, and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service from the underlying hardware. A thin hypervisor can intercept some of those requests and handle them according to some policy.

2.3 Remote Attestation and Initialization

A thin hypervisor can effectively protect the cryptographic key, after it is safely stored in privileged registers and the thin hypervisor is correctly configured. However, the target machine cannot store the cryptographic key prior to hypervisor initialization, and, thus, must obtain it from an external server at some stage.

Obviously, the external server, which we later call the *authentication authority*, has to verify that the remote machine, which is requesting the cryptographic key, is authentic and can protect this key in a potentially hostile environment. This verification process, which we call *remote attestation*, is discussed in chapter 3. Here we will outline the goals of this process.

The cryptographic keys are maintained by an authentication authority, which is equipped with facilities to verify that a thin hypervisor on a remote machine has been properly configured, such that a trusted environment is primed and can accept the cryptographic key. The vehicle to perform this remote verification is a piece of code, called a *challenge* [KJ03, SLS+05, SPvDK04]. The challenge is delivered to the remote machine during the last steps of hypervisor configuration. The remote machine is required to load and execute the challenge code, returning an attestation result to the authentication authority within a limited time-frame. The challenge calculates the checksum of the hypervisor code, but, in addition, mangles the checksum calculation with hardware-driven side effects, sampled by the challenge as it is executing. The authentication authority considers a correct response received within the allotted time-frame proof that the correct hypervisor code is executing and has true control of the remote system's hardware.

The hypervisor's code, being the only verified component, cannot call other functions. In particular, the hypervisor cannot use any of the services provided by the OS. Therefore, the part of the driver which is not included in the hypervisor is responsible for interactions with the OS. Algorithm 2.1 depicts the initialization process of the hypervisor. The process starts by obtaining the challenge code from the authentication authority. This step requires cooperation with the OS. The initialization proceeds by disabling interrupts on all execution units, thus beginning an exclusive execution of the initialization code. At this point we can safely generate sensitive information, which we do by generating random values (used later in the authentication protocol), and storing them in the debug registers. As explained in chapter 3, while one execution unit executes the challenge, other execution units must be suspended. When the challenge execution completes, its result is concatenated with the random value and encrypted using the public key of the authentication authority. Finally, the hypervisor is activated on all execution units and the control is immediately transferred to the guest. Since now the hypervisor can protect the sensitive information stored in the debug registers, we can re-enable interrupts and complete the authentication protocol. The authentication authority encrypts the cryptographic key by the random value, which is stored in the debug registers and protected by the hypervisor. The key is transferred to the driver and then to the hypervisor. The hypervisor decrypts the cryptographic key and stores it in the debug registers (the random value is erased).

2.4 Thin Hypervisor Protection

The two main features of hypervisors that we actively use for protection are interrupt interception and memory virtualization. An interrupt is an event generated by the processor (or received from another processor) that requires immediate attention. The OS can specify a different handler for each such interrupt, by storing the addresses of these handlers in the interrupt descriptor table (IDT). Interrupt interception refers to hypervisor's ability to specify for each interrupt whether the hypervisor should be notified about it before the control is transferred to the interrupt handler. The hypervisor can then decide whether to ignore the interrupt or inject it back to the guest.

There are two approaches to memory virtualization: software and hardware. In software memory virtualization the hypervisor maintains a second hierarchy of virtual page tables, called *shadow page tables*, in addition to the hierarchy constructed by the OS. These shadow page tables are the ones used by the processor, while the OS believes that its hierarchy is used. The hypervisor intercepts all page faults and updates the shadow page tables according to the hierarchy maintained by the OS. Before granting the requested access rights to a page, the hypervisor verifies that this page does not contain sensitive information, which has to be protected by the hypervisor.

Algorithm 2.1 Initialization sequence of the hypervisor. The steps performed by the leading execution unit appear in the left column. The steps performed by all other execution units appear in the right column. The description here assumes that only two execution units are available, but it can be easily extended to any amount of execution units.

Main Execution Unit	**Other Execution Unit**
1. Send info to authority	1.
2. Receive the challenge code	2.
3. Disable interrupts	3. Disable interrupts
4. Generate random value *R*	4. Generate random value *R*
5. Store *R* in debug registers	5. Store *R* in debug registers
6. Suspend other execution units	6.
7. Execute challenge	7.
8. Encrypt challenge result and *R*	8.
9. Resume the next execution unit	9.
10. Suspend current execution unit	10.
11.	11. Execute challenge
12.	12. Encrypt challenge result and *R*
13.	13. Resume all execution units
14. Virtualize	14. Virtualize
15. Return to guest	15. Return to guest
16. Enable interrupts	16. Enable interrupts
17. Send encrypted result and *R*	17.
18. Receive encrypted crypto key	18.
19. Deliver the key to hypervisor	19. Deliver the key to hypervisor
20. Decrypt the key using *R*	20. Decrypt the key using *R*
21. Store the key in debug registers	21. Store the key in debug registers

Hardware memory virtualization allows a hypervisor to define an additional translation hierarchy which maps *guest physical addresses*, i.e. physical addresses as they are perceived by the guest, to actual physical addresses. This additional hierarchy, the extended page table (EPT), allows the hypervisor to specify the access rights of each physical page. When the guest requests rights which are higher than those specified in the EPT, a vm-exit occurs, allowing the hypervisor to decide whether to grant the guest the requested privileges.

2.5 Synchronization

Modern processors consist of multiple execution units. Each unit is almost completely autonomous, which means that each unit executes a separate instance of a hypervisor that share (but do not have to, in general) the code and some data structures. As we will see in chapter 4, sometimes one instance of the hypervisor has to communicate with another instance of the hypervisor. Sometimes one instance even wants to request that another instance perform some operation. This is a classic problem, which has two general solutions: a synchronous and an asynchronous one.

The asynchronous solution suggests maintaining a data structure for each instance. Each instance periodically inspects this data structure and services all requests recorded by this data structure. Other instances can submit their requests by writing them to this data structure. Hypervisor's configuration allows the specification of a maximal amount of cycles after which a vm-exit is guaranteed to happen. This mechanism can be used to perform some periodic operation in the hypervisor.

In the synchronous solution, one execution unit notifies another execution unit by sending an inter-processor interrupt (IPI). When sent, an IPI causes the destination execution unit to preempt its current execution and jump to an interrupt handler. This solution is usually much more efficient, since it shortens the request's response time. Unfortunately, it is also much more complex, since it involves modification of OS internal data structures. On 64-bit Windows, such modifications are usually detected by the PatchGuard and cause the computer to restart.

3 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 [KJ03, SLS+05, YHL+11, SPvDK04, CFPS09, SWP08, SLP+06, YWZC07]. 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 [Ion09, Son15, Bri15], and only authenticated bank terminals can be allowed to fetch records from the bank database [Wik15].

3.1 Previous Work

The research in this area can be divided into two major branches: hardware-assisted authentication and software-only authentication. Whilst, in theory, hardware-assisted authentication may provide more conclusive results regarding the authenticity of a remote machine, in practice the hardware fails to provide additional security due to the inappropriate designs of currently available operating systems [SWP08].

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

Usually [Pea02, ELM+03, SZJvD04] the TPM is used as the root of the chain of trust. The TPM measures the authenticity of the BIOS. The BIOS then measures the authenticity of the boot loader and so on. Unfortunately, all common modern operating systems (e.g. Linux, Windows, OS X) allow the user to load drivers for execution with the same privileges as the operating system itself, i.e. ring 0 on x86 and x64 hardware. Malicious or buggy drivers, which are executed with high privileges, allow random code execution that makes it possible to circumvent the authenticity measurements of the TPM.

Software-only authentication usually targets a specific instruction set architecture that varies from ATMega [SPvDK04], through Pentium [KJ03] to Intel Core [YHL+11]. 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.

3.2 Our Method

Kennell and Jamieson proposed [KJ03] 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.

This work extends the method of Kennel and Jamieson, and adapts it to modern processors. A detailed discussion of all the extensions can be found in [PI, PIII]. Figure 5 depicts the interaction between the authentication authority and the target computer. 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 target's authenticity. The authentication authority encrypts the secret information with the random value from message (3) acting as an encryption key, and transmits the encrypted

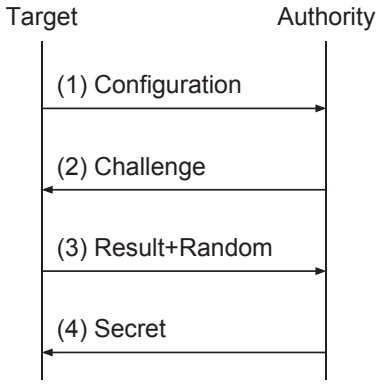

FIGURE 5 The attestation protocol between the authentication authority and the target machine. The protocol consists of four messages. The first two messages are sent unencrypted, while the two last messages are encrypted. The third message is encrypted by the public key of the authentication authority and the fourth message is encrypted by the random value transmitted in the third message.

message to the target machine.

During every communication session, the authentication authority selects a random challenge from the set of challenges that suit the target's configuration. During the construction of the challenge repository, the authentication authority attaches additional information to each challenge — for example, the expected result of the challenge and the maximal amount of time needed for its execution. The authentication authority denies target machines whose results differ from the expected, or were reported outside the predefined time-frame.

The population of the challenge repository is a randomized iterative process, which generates a challenge (the randomized part), computes its result, and stores the information in the repository.

Figure 6 presents the general structure of a challenge, which mostly consists of checksumming and side effect accumulating blocks. Two special blocks that appear in all challenges are the prologue and the epilogue. The prologue block is the entry point of the challenge. This block is responsible for setting up an appropriate execution environment for the challenge (explained below). The prologue then proceeds by initializing the result variable (actually, register) to zero and transferring control to another block: a side effect accumulating block or a checksumming block. A side effect accumulating block retrieves information about the current internal state of the processors and incorporates it into the result variable. Then, depending on the value of the result, it jumps to one of the three predefined blocks. A checksumming block advances (in a pseudo-random fashion) a memory pointer to a new location, reads the content at this location and incorporates it into the result. If the final location was reached, the checksumming block jumps to the epilogue block, otherwise it jumps to one of the three predefined blocks

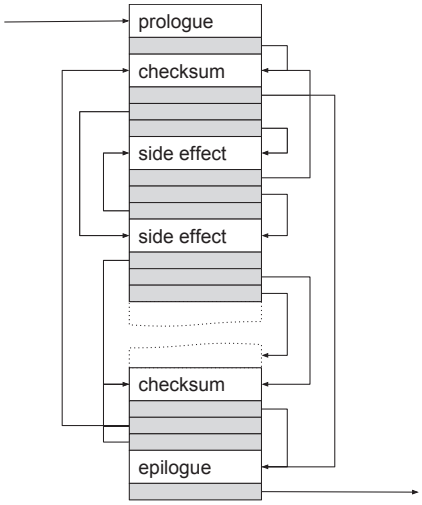

FIGURE 6 The structure of the attestation challenge. The white rectangles, together with the gray rectangles beneath them, represent the challenge blocks. Each gray rectangle is a conditional control transfer, which is represented by an arrow to another block. The white rectangles are the blocks themselves, which compute checksum or incorporate side effects.

based on the result. The epilogue block restores the execution environment and returns the computed result to the caller.

3.3 Side Effects

Every instruction that is executed by a processor modifies its internal state. Some modifications result from the definition of the instruction operations; others are performed by the processor to improve performance, e.g., cache population; or for debugging and profiling purposes, e.g., L3 cache miss count. Previously, processors were allowed to observe the state of side effects directly. Current versions of processors provide a different mechanism — performance counters. The processor defines pairs of registers: an event selection register, which allows the software to specify the execution event to be counted, and a monitoring counter register, which is increased on each occurrence of the event specified by the first register. The values of the counter registers can be considered the state of the side effect and as such can be incorporated into the result by the side effect accumulating blocks.

It is desirable to construct the challenge in a way that maximizes the side effects produced by its execution. One of the side effects that were considered in [KJ03] is the TLB management system. TLBs store translations of virtual addresses to physical addresses of pages that were recently accessed. Modern pro-

cessors contain separate TLBs for instructions and data as well as a shared TLB of a higher level, which is larger but slower. When a new translation needs to be stored in a TLB with no free slots, one of the slots is evicted according to some policy, which varies between processors. In order to achieve high utilization of the TLBs the authors of [KJ03] propose to map a large virtual memory region that maps a smaller physical memory region that is to be authenticated. The challenge then can compute the hash by reading the contents of the physical memory region through different pages of the virtual memory region, thus fully utilizing the DTLB and inducing more side effects. In order to fully utilize the ITLB, we map the challenge blocks to different virtual pages and use their addresses in those pages as the control transfer destinations.

As we have seen in chapter 2, a hypervisor can prevent the virtual environment from accessing some physical pages. Moreover, the virtual environment will be unaware of this situation. In other words, a hypervisor can deceive the virtual environment into thinking that it has full control of the underlying hardware and no other software is currently being executed. Obviously a challenge, being part of the virtual environment, can be deceived as well.

Fortunately, some events are intercepted by hypervisors unconditionally. In particular, on processors manufactured by Intel, execution of the CPUID instruction always causes a vm-exit. On vm-exit, the processor loads the first instruction of the function whose address is specified in VMCS. This behavior alone will affect some of the caches, regardless of the actual implementation of the function. The lookup of the address modifies at least one entry of the ITLB and the higher level TLB (STLB). Fetching the first instruction modifies at least one entry in the instruction cache, L2 cache and L3 cache. In addition, execution of such an instruction takes much more time when a VMM is active. Therefore, in challenges, which are targeted at modern processors, we widen the variety of blocks by adding blocks that produce events whose interception cannot be disabled. An example of such a block is a block that contains a CPUID instruction.

In modern processors, the number of possible performance events greatly outnumber the available hardware counter circuits. Most processor models are restricted to 2-4 individual performance counters. Therefore, it is possible to dynamically link an available performance counter to a specific performance event. Once linked, the performance counter counts the number of events that occurred.

One of the challenge's goals is to determine if the remote machine is executing under emulation or not. Two factors are measured to determine this: the challenge result and the challenge's elapsed execution time. Since the result of a challenge is affected by the values of performance counters at different execution points, an emulator is forced to keep track of their values. Even assuming that such a feat is possible with regard to one of the side effect modules, referencing several modules in a single challenge would necessarily amplify the elapsed execution time differences, since these emulations are mostly orthogonal.

It is, therefore, desirable to utilize a large variety of performance measurements. Each such performance measurement increases the execution time of an emulator, but has no effect on a non-emulated system. A clear deficiency with

respect to this is the low ratio of available performance counters to possible performance events that can be measured. We suggest to overcome this deficiency by using *chained performance-counters*.

The idea is to monitor many side effect inducing modules with a much smaller number of available performance counters, by shifting the counters from one module to the next according to a set of deterministic rules. All processor components that generate side effects are initialized to a known state before the challenge execution begins. When challenge execution flow reaches a determinable point, the contents of each side effect inducing module is deterministic and repeatable regardless of our measurement, i.e whether a performance counter was used to monitor its side effects or not. It follows that a performance counter can be connected to the module to count new events. The new events will occur deterministically for the active challenge given the new determinable state.

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

4 NATIVE CODE EXECUTION

Despite the rising popularity of managed execution environments, such as Java and .NET, native programming languages, such as C and C++, are still widely used, especially in areas that require high performance, e.g. Web browsing, image and video editing, gaming, etc. Many of these programs contain proprietary algorithms and licensing schemes, which might be subject to reverse engineering and modification. The main countermeasure which is currently available is obfuscation. As discussed in chapter 1 this measure can be easily circumvented by a skilled professional and is, therefore, unreliable.

We propose to protect the sensitive parts of a native program by encrypting them. Encryption and execution of encrypted code require special software components. This chapter summarizes the detailed description of these components provided by [PVII]. The relationship between the components is depicted in Figure 7. The general purpose and the abilities of a hypervisor, as well as the necessity for a driver wrapper, were discussed in chapter 2. The attestation software was discussed in chapter 3.

4.1 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. Figure 8 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

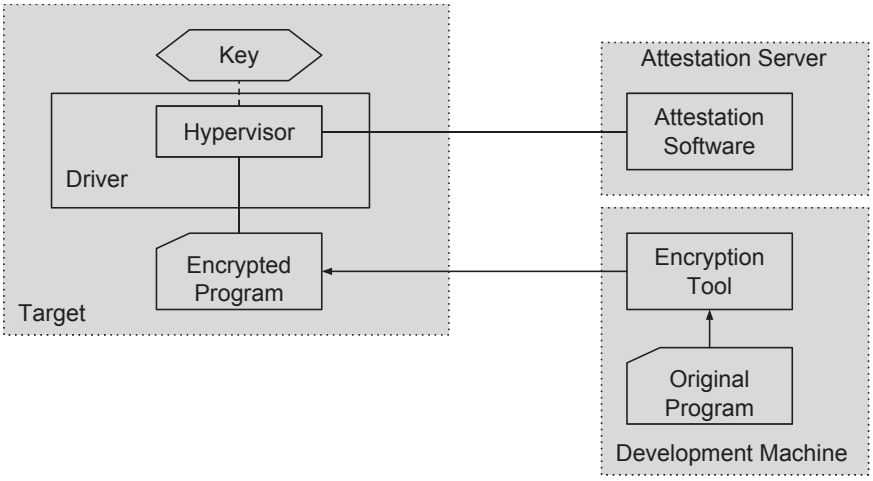

FIGURE 7 Native code protection system. The original program is encrypted before its distribution. The encryption key is stored in the attestation server, which delivers it to the hypervisor in the target machine upon successful attestation. The hypervisor is initialized by a driver, which also hosts the code of the hypervisor.

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.

The encryption tool modifies the given PE 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

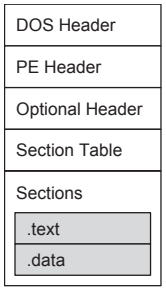

FIGURE 8 Structure of a Windows PE file. The structure contains a variable number of sections. Two of the most common sections are presented.

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.

As will be explained in section 4.2.1, 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 lies outside the function's bounds, or if the destination cannot be determined statically (if it is 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 9 presents an example of such transformation.

4.2 Encrypted Program Execution

In order to execute an encrypted program, the user must first install the driver, which wraps the hypervisor. The driver monitors the PE files loaded by the OS, and keeps track of PE files that contain the special 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 verification, obtains the cryptographic key, and enters a virtualized state. Chapter 2 provides a detailed description of this process.

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, 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 driver 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

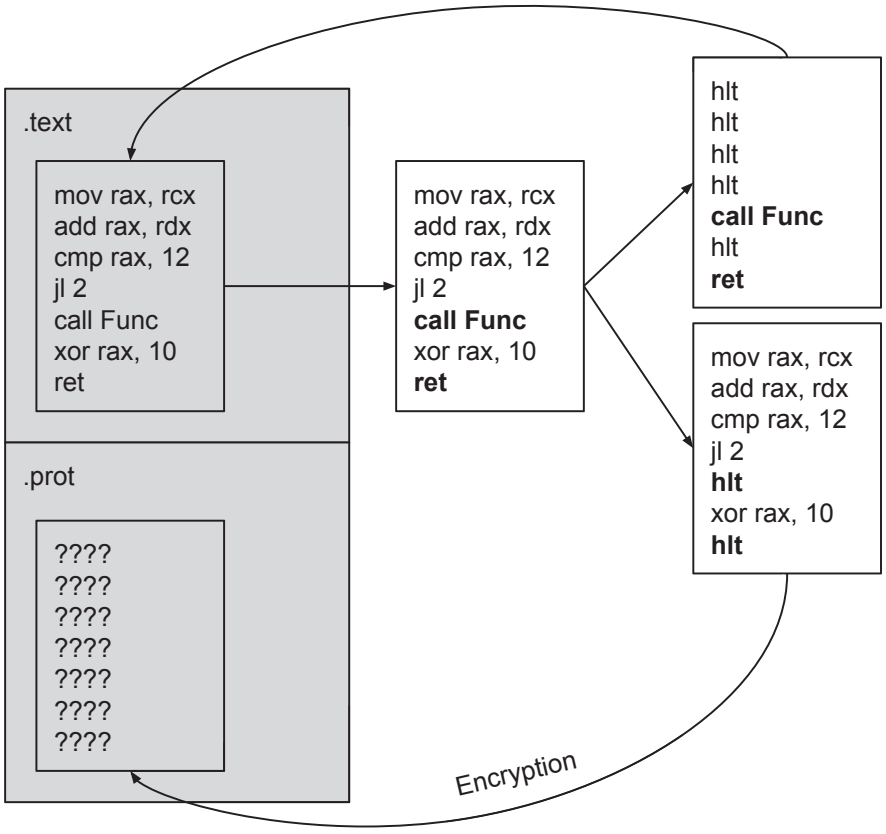

FIGURE 9 Example of an encryption process of a single function. The encryption begins by classifying instruction is encryptable (normal face) and non-encryptable (bold face), and creating to copies. The complementary instructions in each copy are replaced by halts. Finally, one copy is written over the original functions, and the other is encrypted and added to the special section.

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 unencrypted 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.

4.2.1 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 chapter 2. The hypervisor switches to the hot state when the memory contains sensitive information, which cannot be protected by a regular memory protection technique (using EPT), since its physical location is not known (or not constant). This switch occurs when the hypervisor starts execution of an encrypted function.

In the following description, we assume that the encryption tool uses halt as a replacement instruction, but the same is true for the software breakpoint instruction.

At the initialization the hypervisor's state is set to cold. In this state, in addition to the regular protection means described in chapter 2, the hypervisor intercepts general protection faults. An encrypted function which was overwritten by the NEC consists mainly of halt instructions. Execution 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 the NEC. Then the hypervisor switches to hot mode and decrypts the EC to its natural location, currently occupied by the NEC (the NEC is copied to 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 behaviour 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.

When a vm-exit occurs in hot mode, the hypervisor switches to cold mode. First, the decryption function is replaced by the corresponding NEC. Then the hypervisor unfreezes all the execution units, configures itself to intercept only general protection faults, and returns to the guest. Figure 10 depicts the control flow during encrypted function execution.

We suggest freezing other execution units by inducing a vm-exit on each execution unit, and running a busy loop until the hypervisor switches back to

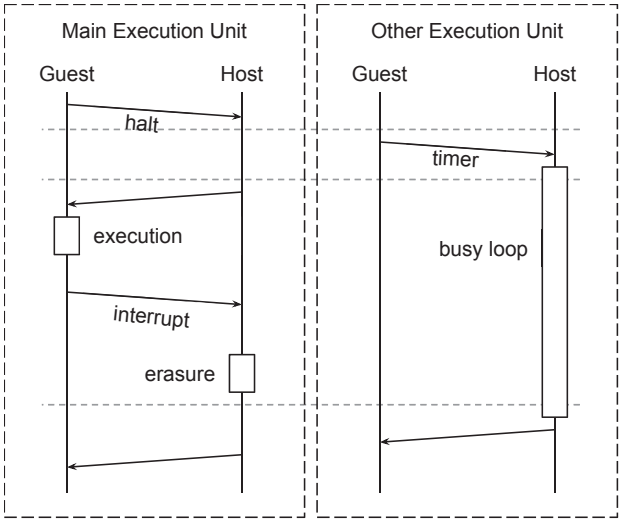

FIGURE 10 Example of encrypted function execution. The figure depicts two execution units, each with two alternating states: guest and host.The dashed horizontal lines are synchronization barriers, i.e. everything above the line is guaranteed to complete before anything below the line starts.

cold mode. A vm-exit can be induced either implicitly on a timer, or explicitly by sending an inter-processor interrupt (IPI). The former solution is much easier to implement but the latter 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 moving the instruction pointer of the guest to point to the Nth interrupt handler of the original IDT.

4.2.2 Buffered Execution

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. As a consequence these functions have the same privileges as the hypervisor itself. 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 a random code, without knowing the cryptographic key. Unfortunately, it is possible that some memory manipulation can be performed indirectly by modifying the data on which the encrypted function works. Although possible, it seems to be extremely difficult to manipulate the behaviour of an unknown code through its data.

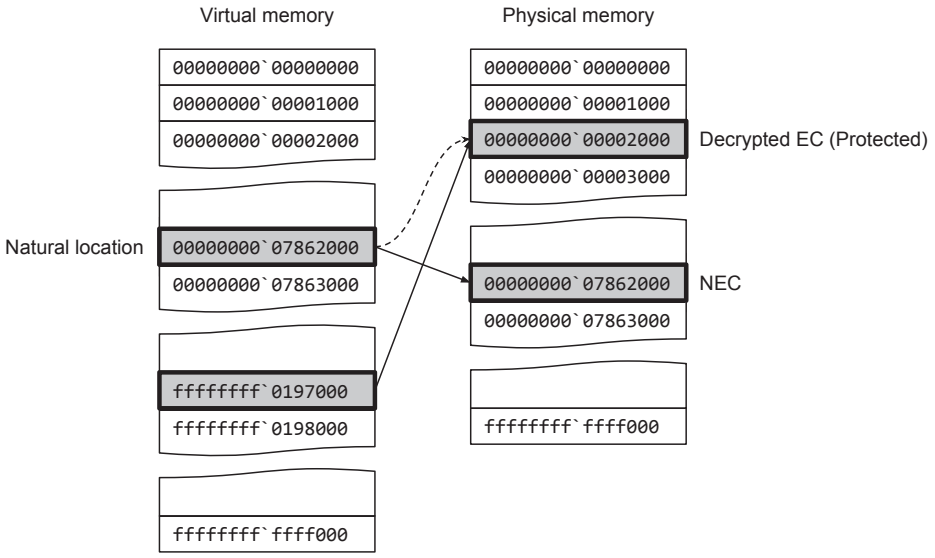

FIGURE 11 Memory layout during buffered execution. The functions resided at virtual address 7862000, which is mapped to the physical address 7862000 (a coincidence). The encrypted code is decrypted to virtual address ffffffff'0197000 which is mapped to the physical address 2000. The hypervisor changes the mapping of the virtual address 7862000 to map the physical address 2000.

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.

In this approach, the hypervisor has only one state, in which it protects itself as described in chapter 2. 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 to a pre-allocated memory buffer, which is protected by the hypervisor. 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.

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 that 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 from 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 pre-allocated buffer containing the decrypted EC. Figure 11 depicts this transformation.

The hypervisor proceeds by restoring the registers of the guest and transferring control to the decrypted EC. The decrypted EC executes inside the hypervisor (in kernel mode) until it reaches a software breakpoint instruction. At this point, the processor invokes the breakpoint trap handler, which saves the registers and returns to the guest. Other interrupts, such as page faults, which can occur during a normal course of program's execution, are handled in a similar fashion.

4.3 Comparison

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 12.

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 ??); this mode is never used by the system described in this paper. In this mode we can prevent attempts to execute privileged instructions or access 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 modi-

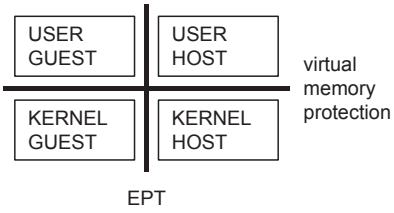

FIGURE 12 Execution modes. The left column represents the guest mode, while the right column represents the host mode. The lower row represents the kernel mode, while the upper row represents the user mode. The host mode can protect itself from the guest mode through the EPT mechanism. The kernel mode can protect itself from the user mode through the virtual memory protection mechanism.

fies 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.

5 JAVA CODE EXECUTION

Executing encrypted managed program, such as Java programs, is more challenging than executing native code, and this results in a more complex system design, which is described in detail in [PVI] The design is further complicated by our desire to provide near-optimal execution performance for the unencrypted parts of a Java program. In order to achieve near-optimal performance, we must cooperate with an industrial-grade JVM. Fortunately, the Java standard specifies two interfaces, JNI and JVM TI, that allow the realization of such cooperation.

5.1 JVM TI

JVM TI [Ora] is an application programming interface (API) provided by the JVM that allows the inspection and control of the state of the JVM and the program it executes. This API is usually used performance profilers and debuggers [BH06, HAD10, Lon05, Lue12]. JVM TI is a two-way interface. A client of JVM TI, an *agent*, can be notified of interesting occurrences through events. An agent can query the JVM through many functions, either in response to events or independent of them. An agent is realized as a dynamic library. The JVM loads the agent during initialization and allows it to specify a list of events to be intercepted, e.g., class loading, method invocation, exception handling, etc. When an event occurs, the JVM invokes the interception function of the agent associated with this event.

In addition to events interception, JVM TI allows an agent to inspect and manipulate the state of the JVM and the state of the program it executes. One family of functions allows the inspection of the program structure. For example, the GetClassMethods function retrieves the method identifiers of a specified class, and the function GetMethodName retrieves the name and the signature of a specified method. Another family of JVM TI functions allows the inspection of dynamic aspects of the execution. This family includes functions such as GetLocalVariable, which retrieves the value of the method's local variable (in Java parameters are also variables), and GetStackTrace, which retrieves informa-

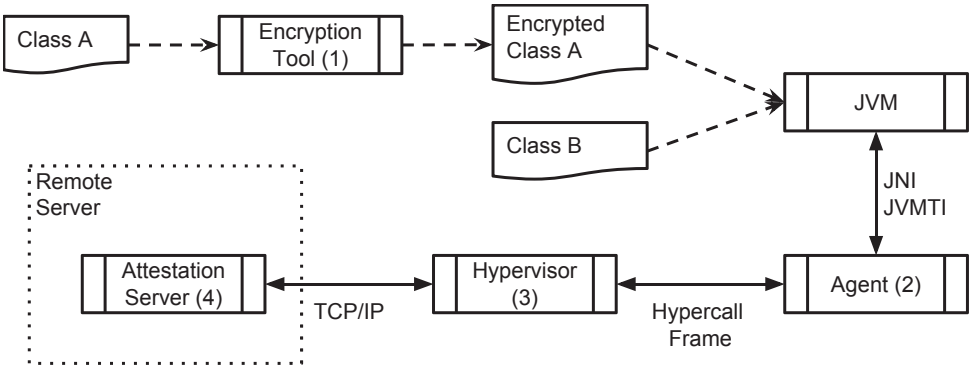

FIGURE 13 Relationship between the different components of the described system. The encryption tool (1) transforms regular Java classes into encrypted ones. Regular and encrypted Java classes are then loaded by the JVM. The JVM loads a JVM TI agent (2) through a JVM TI interface. The agent links the hypervisor to the JVM and assists in the interpretation process. The agent communicates with the JVM through JVM TI and JNI. The communication between the agent and the hypervisor is based on hypercalls and execution frames. The hypervisor receives the decryption key from a remote server, which attests the validity of the hypervisor and the hardware on which it executes.

tion about the callers of the current method. Finally, another family of JVM TI functions allows for the modifying of the program's state. This family includes functions such as SetLocalVariable, which assigns a value to the method's local variable; SetBreakPoint, which sets a breakpoint at a specified location of a specified method; and ForceEarlyReturnObject, which requests to terminate the execution of the current method.

5.2 System Design

The system we present comprises four main components: (1) encryption tool, (2) JVM TI agent, (3) thin hypervisor, and (4) attestation server. Figure 13 depicts the relationship between these components.

The encryption tool processes each class file (see Figure. 14), an execution unit of a Java program, by first de-serializing it into memory based structures. The code bytes of each method are located and zeroed out to create a sequence of NOP instructions. The encryption tool extends the existing Constant Pool to make room for encrypted versions of protected methods' bytecode. The original bytecodes of each method are encrypted and inserted in a new record appended at the end of the Constant Pool table.

During the initialization of the JVM TI agent, it deploys the hypervisor and

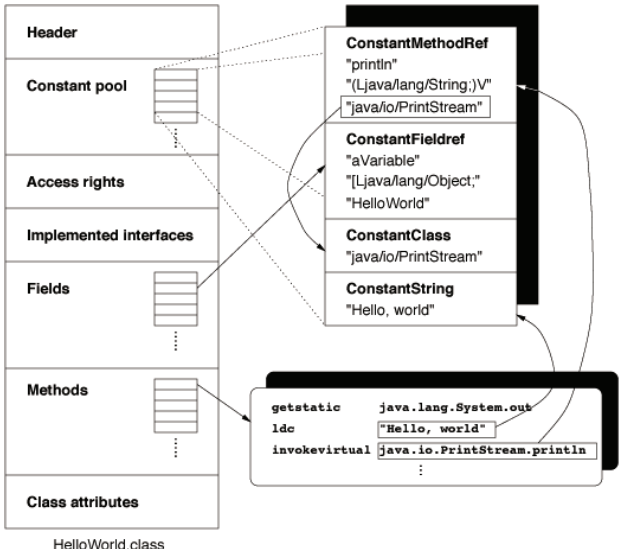

FIGURE 14 Structure of Java's class file. The constant pool contains all the constant referenced by the class as well as identifiers of all referenced classes, methods and fields. Below the constant pool reside the definitions of the fields and the methods of the class. Each method contains the bytecode of its implementation.

installs interception functions for the class loading and the breakpoint events. The class loading event occurs whenever the JVM loads a class and before any of the class code is executed. Upon this event the agent inspects the class and determines whether it is encrypted. If so, the agent installs a breakpoint at the first instruction of each method. These breakpoints induce a breakpoint event on each entry to the encrypted methods. The agent intercepts the breakpoint event, resolves the method that hosts the hit breakpoint, and begins the interpretation process.

The interpreter constructs a frame, a data structure which constitutes the execution environment of the current method invocation (including the encrypted bytecodes of the current method), and transfers control to the hypervisor. The hypervisor decrypts the bytecodes and starts interpreting them one-by-one until it reaches an opcode which requires cooperation with the JVM. At this point, the hypervisor returns control to the agent and provides it with the instruction, which it could not interpret, in decrypted form. The agent proceeds by interpreting the instruction using JVM TI and JNI and then transfers control back to the hypervisor. Figure 15 presents the control flow diagram of the system operation.

The interpretation is performed by two interpreters: one is embedded in the JVM TI agent and the other is embedded in the hypervisor. Each opcode is interpreted by only one of the two interpreters. When one interpreter cannot continue interpretation, it transfers the control to the other interpreter. The interpreters share a data structure, which we call a frame, in which they store the intermedi-

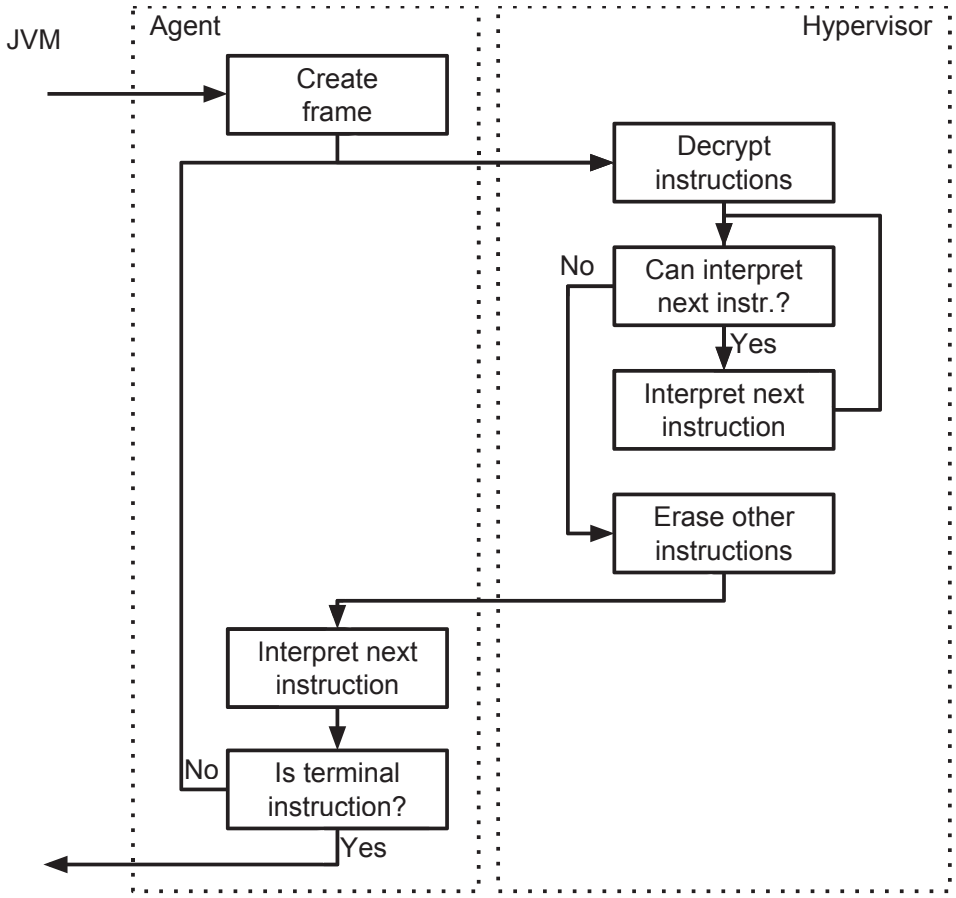

FIGURE 15 A simplified control flow during encrypted method execution. The JVM reaches the breakpoint installed by the agent, and transfers the control to the JVM TI agent. The agent creates a frame and transfers the control to the hypervisor. The hypervisor decrypts the instructions, and interprets them until an uninterpretable instruction is reached . Then, the hypervisor erases all the other instructions and returns control to the JVM TI agent, which interprets the instruction and either transfers the control to the hypervisor or returns control back to the JVM.

ate results of the interpretation as well as some additional information.

We want to enable the interpreter, which is embedded in the hypervisor, to interpret as many instructions as possible. Many instructions — arithmetic, logic, type conversion, stack management, control transfer — operate only on the stack and the program counter (PC), and can be easily interpreted by the hypervisor. The load and store group of instructions allow the program to push the value of local variables and constants onto the stack. In order to enable the hypervisor to interpret instruction in these groups as well, we include the constant pool and the local variables in the frame.

Some instructions, nevertheless, cannot be interpreted by the hypervisor. Instructions that are responsible for object creation and manipulation, or method invocation and return, require cooperation with the JVM. For example, the *getfield* instruction, which pushes onto the stack the value of the specified field in the specified object, must inspect the internal representation of the object as defined by the JVM. Another example is the *return* instruction, which terminates execution of the current method. This instruction must modify the internal representation of the stack trace, which is managed by the JVM. Therefore, all these instructions are interpreted by the JVM TI agent via JNI and JVM TI functions.

When the hypervisor encounters an instruction that requires cooperation with the JVM, it delivers it to the JVM TI agent for interpretation. The JVM TI agent interprets the delivered instruction by invoking the appropriate JVM TI and JNI functions.

5.3 Measurements

According to [CMS07], *invokevirtual* is the second most popular instruction (appears with 8.9% frequency) and *getfield* is the fourth most popular instruction (5.4%). Unfortunately, these instruction cannot be interpreted inside the hypervisor, and therefore, they are delivered in a decrypted form to the JVM TI agent, which is not considered secure. According to the statistics in Table 1, the hypervisor delivers about 38% of the instructions in a decrypted form back to the JVM TI. Therefore, in practice, only about 60% of the instructions in an encrypted class are actually hidden from an adversary.

As we have seen, Java programs can be, at least partially, protected from an adversary. We believe that this degree of protection is sufficient in cases where traditionally obfuscation was used. In other cases, which require a higher degree of protection, we suggest either avoiding using uninterpretable (by the hypervisor) instructions, or use a tool which can reduce the frequency of uninterpretable instruction, for instance, by inlining methods.

TABLE 1 Frequencies of uninterpretable instructions as reported by [CMS07]

invokevirtual	8.9	invokeinterface	1.1	iaload	0.3
getfield	5.4	iastore	1.1	newarray	0.2
invokespecial	3.8	ireturn	1	instanceof	0.2
new	2.5	checkcast	0.7	ifacmpne	0.2
putfield	2.1	bastore	0.6	sastore	0.1
invokestatic	1.7	athrow	0.6	monitorexit	0.1
return	1.6	putstatic	0.4	lastore	0.1
getstatic	1.6	anewarray	0.4	castore	0.1
areturn	1.3	aaload	0.4	baload	0.1
aastore	1.2	arraylength	0.3	**Total:**	**38.1**

6 CONCLUSIONS

In this chapter, we summarize the contributions of the thesis, discuss the limitations of the reported research, and outline some directions for further research.

6.1 Contribution

The main contribution of the thesis is a complete system for execution of encrypted programs. The design of this system combines several components, some of which were previously studied, and others which are novel. The thesis describes the novel components and extends the studied components. The contribution of this thesis is, therefore, not limited to the area of encrypted code execution.

The thesis extends the remote authentication method, which was studied by Kennell and Jamieson [KJ03] and others [SLS+05, SPvDK04, SLS+05, SLP+06]. The extension adapts this method to modern processors, which have multiple execution units, a wide variety of performance events, and hardware-assisted virtualization.

The thesis extends the notion of a thin hypervisor, proposed by [SET+09, CSK10], to code protection. A complete system is described, which includes remote authentication, cryptographic key delivery, and secure execution of encrypted methods. This work presents two execution techniques which vary in their security and performance, thus allowing an engineer to apply the appropriate technique in each case.

Finally, the thesis extends the idea of encrypted program execution to managed programs. In particular, we present a complete system for encryption and execution of Java programs. The system introduces the idea of co-interpretation — interpretation of one program by a secure and a non-secure interpreters. This work presents an evaluation of the performance and the security of the system.

6.2 Limitations and Further Research

This section presents the limitations of the described systems and outlines directions of further research.

6.2.1 Native Programs

Chapter 4 discussed two methods of native code execution: in-place execution and buffered execution. The advantages and disadvantages of each method were presented and compared. While the buffered execution method has a better performance, the in-place execution is more secure. An engineer can decide which method should be applied for a given problem.

The chapter presents an additional memory protection mechanism, which can be used to provide the security of the in-place execution method and the performance of the buffered execution method. We plan to research this approach in the near future in order to improve the overall performance of the system.

6.2.2 Java Programs

As reported in chapter 5, the performance and the security of the Java system are not optimal. In particular, the performance can be improved, by caching the state of objects in the hypervisor, thus eliminating the need to transfer control back to the JVM TI agent on every invocation of the *getfield* and *setfield* instructions.

Another family of costly instructions is a family of method call instructions. We can eliminate the control transfer to the JVM TI agent, when an encrypted method calls another encrypted method, by emulating the call stack manipulations and the argument passing, which are usually performed by the JVM. This emulation not only saves the redundant vm-entry and vm-exit, but also eliminates the need to call the costly JVM TI functions that retrieve the method's arguments values.

Finally, there are many other managed execution environments, such as .NET, Javascript, Python, etc. To the best of our knowledge, these environments do not offer an equivalent of JVM TI. Therefore, these environments will probably require to embed the interpreter, which is now part of the JVM TI agent, in the execution environment itself. While being more complex, this solution potentially also seems to be much more efficient, since both such an embedded interpreter, as well as the hypervisor's interpreter, are aware of the internal representation of the objects and other data structures, thus allowing them to interpret much more instructions in the hypervisor.

7 SUMMARY OF ORIGINAL ARTICLES

In total 7 articles are included in the thesis. This chapter provides the summary of each. For each article, the addressed research problem is formulated first, followed by the description of the obtained results.

7.1 Trusted Computing and DRM

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

7.1.1 Research Problem

This book chapter surveys novel trusted computing solutions based on the Trusted Platform Module (TPM) as advertised by the Trusted Computing Group. We discuss the working principles and programming interface of the TPM. The chapter covers the following aspects of the TPM: protected memory, remote attestation, direct annonymous attestation, measurement counters and key storage.

7.1.2 Results

The TPM offers a key storage and an execution context, which are protected by hardware obfuscations. While hardware-based security, as a result of its design, is not vulnerable to eavesdropping or side-channel attacks, it has two main deficiencies. The first deficiency is the requirement of additional hardware, which can increase the cost of the final product. The second deficiency is the lack of flexibility. While the TPM allows the user to attest an application and perform some cryptographic primitives, it does not support execution of encrypted programs. The approach which is described in this chapter is complementary to the system presented in this thesis.

7.2 An Efficient VM-based Software Protection

Averbuch, A.; Kiperberg, M.; Zaidenberg, N.J., "An Efficient VM-based Software Protection" in *Network and System Security (NSS), 2011 5th International Conference*, pp.121-128, 6-8 Sept. 2011

7.2.1 Research Problem

This article studies the obfuscation methods for software protection, which are predominantly used for digital rights management (DRM). The most sophisticated obfuscations translate a program from one ISA to another, thus rendering a familiar x86 code incomprehensible to a human reader. Obviously, regular processors cannot execute such programs directly, thus forcing the developers of such obfuscation methods to equip the transformed program with an interpreter of the new ISA. Researchers showed that the presence of the interpreter reveals enough information about the underlying ISA to reconstruct the original program.

A completely different approach was proposed by [Bes80]. According to this approach a cryptographic key is burned into every manufactured processor. A program, which is intended to be executed by such a processor, is encrypted with the key of this processor. The program is copied to the target computer in its encrypted form. The processors executes this program by reading its instructions one-by-one and decrypting them inside the execution pipeline. This article investigates the viability of achieving a comparable level of security on regular processors.

7.2.2 Results

The article introduces a new framework for a protected execution of code. This framework comprises of an encryption tool and an execution environment. The encryption tool acts as an obfuscating transformation — it translates the program from the original ISA to a new ISA and encrypts the instructions using some key. The execution environment decrypts and executes the instructions.

The decryption key is stored in a remote server. The remote server provides the execution environment with the decryption key upon successful validation. The purpose of the validation process is to ensure that the key is provided to a legitimate entity (the real execution environment), which is able to store the key securely.

The article suggests the use of a polyalphabetic cipher for program encryption and describes an efficient method of its realization. Polyalphabetic ciphers are very efficient but less secure than “real” ciphers, like AES or 3DES. However, in many cases, we can sacrifice security for performance. The availability of an alternative makes it an engineering decision. The article evaluates the performance of the different encryption schemes.

7.3 Truly-Protect: An Efficient VM-Based Software Protection

Averbuch, A.; Kiperberg, M.; Zaidenberg, N.J., "Truly-Protect: An Efficient VM-Based Software Protection" in *Systems Journal, IEEE*, vol.7, no.3, pp.455-466, Sept. 2013

7.3.1 Research Problem

This article extends the research published in [PV]. The article summarizes the different approaches for defying reverse engineering and means to circumvent them. All the reverse engineering prevention systems are based on obfuscation, which is vulnerable to hacking or user malicious activities. Therefore, it is safe to conclude that software should be protected by encryption rather than obfuscation.

7.3.2 Results

The article presents a detailed description of a system for encrypting and executing native programs. The article claims that a polyalphabetic cipher, being extremely fast, can be used to encrypt moderately sensitive programs. The performance of different ciphers is studied and reported, to enable an engineer to make an informed decision regarding the encryption method.

The article presents a slight modification to the original design of the system, that allows it to achieve a better performance on processors with multiple execution units. The performance boost is achieve by parallelizing the interpretation; instructions decryption is performed by one execution unit while decrypted instructions execution is performed by another execution unit.

7.4 Efficient Remote Authentication

Kiperberg, M.; Zaidenberg, N.J., "Efficient Remote Authentication" in *The Journal of Information Warfare*, vol.12, no.3, Oct. 2013

7.4.1 Research Problem

Remote computer authentication is a process by which one computer ensures that another computer is trustworthy. The exact properties of this trustworthiness depend on the application, but generally the computer under test has to prove that it is an actual, non-virtual, computer executing a legitimate software.

A method was proposed [KJ03] for establishing trustworthiness of a remote computer, in which the authority sends a challenge code to the computer under test and verifies that the correct result was received within a predefined period of time. The challenge code has a special design, which prevents the adversary

from achieving the correct result within the time constraints. The two main components of the challenge code are checksumming and side effect accumulation. The checksumming component computes a hash of the memory region containing the software under test. The side effect accumulation component guarantees that the computation cannot be simulated efficiently enough to accommodate the time constraints.

The originally proposed method [KJ03] assumes that the software under test is known in advance. While this assumption may hold in some embedded devices, it definitely fails on personal computers or even complex portable devices, such as phones and players. This article studies the structure of the software in these devices and extends the proposed method to these dynamic environments.

7.4.2 Results

Popular operating systems have tens and hundreds of versions. In addition, the device drivers are executed with the same privileges as the kernel itself, thus forcing the authority to verify them as well. This article claims that the amount of device drivers increases very slowly over time and it is feasible to track them. Moreover, only a small fraction of the drivers are used in practice, which decreases the number of possible configurations of the system.

The article presents an automatic system that pre-computes the correct results of the challenge codes, as described by Kennell and Jamieson. In addition the system schedules new computation requests for remote computers with previously unseen configurations. This computation request is serviced automatically if the new configuration consists solely of authorized device drivers. Otherwise, a manual intervention is required.

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

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

7.5.1 Research Problem

The problem of remote software authentication is determining whether a remote computer system is running the correct version of a software. 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 [Ion09, Son15, Bri15] and only authenticated bank terminals can be allowed to fetch records from the bank database [Wik15].

The research in this area can be divided into two major branches: hardware assisted authentication and software-only authentication. This article concentrates on software-only authentication, in which the authentication authority sends a challenge code to the computer under test and verifies that the correct result was received within a predefined period of time. In particular, this article extends a previously known method of software-only authentication [KJ03] to modern machines. Modern machines have two main features that require a special consideration by the challenge code designer.

The first feature is virtualization, which is available on processors designed by Intel [Int07] and AMD [AMD10]. Virtualization allows a potential adversary implementing an emulator which is as efficient as the original machine, thus defeating the authentication scheme.

The second feature is multi-processing, which is also widely implemented by different manufacturers. The authentication method, which is referenced by this article, accumulates information about side effects of the computation. Unfortunately, some of the processor's internal state is shared by the different execution units (logical processors). Therefore, one execution unit can affect the computation of another execution unit, resulting in an incorrect computation and eventually failing the authentication procedure.

7.5.2 Results

The article presents an extension to the original authentication procedure. This extension addresses the two features of modern processors.

Virtualization on modern processors enables the system software to deliberately intercept execution of privileged instructions and access the hardware resources. Interception of some instructions may depend on the processor's configuration while other instructions are intercepted unconditionally. One such instruction is CPUID, which is generally used to inspect processor's features, and is guaranteed to not affect the memory. However, when the virtualization is enabled, CPUID causes the system software to intercept this instruction, thus affecting the memory. The article suggest incorporating the CPUID instruction in the challenge code.

The multiplicity of execution units is resolved by stalling all logical processors but the one that is currently executing the challenge code. The article suggest to implement the stall using a MONITOR/MWAIT pair of instructions. These instructions take a specified memory range and put the processor in an idle state until the contents of that specified memory region is modified. Since no instructions are executed, other execution units are not affected by the idle logical processors.

7.6 System for Executing Encrypted Java Programs

Kiperberg, M.; Resh, A.; Algawi, A.; Zaidenberg, N.J., "System for Executing Encrypted Java Programs" in *IEEE Transactions on Dependable and Secure Computing*, Submitted

7.6.1 Research Problem

This article studies the applicability of secure execution of encrypted (native) program to managed program. Unlike regular (native) programs, managed programs cannot be executed directly by the CPU and therefore require a special (native) program to interpret the managed program. Managed execution environments are superior to native environments in memory management, debugging and profiling support. These factors have caused them to become popular among developers of desktop and mobile applications.

While it is possible to guarantee that a sequence of native instructions cannot be intercepted (read or modified) during its execution by a CPU, such guarantees cannot be made for a managed execution environment, since an unexpected behavior can be introduced into the software that implements the managed execution environment. There is, therefore, a need for a technique for executing safely encrypted managed programs on the available managed execution environments.

Java is the most popular example of a managed execution environment. Java's specifications are publicly available which makes it a good candidate for a use-case study. Moreover, Java provides standardized interfaces that allow to inspect and modify the internal state of the program and the execution environment. These interfaces, called JVM TI and JNI, allow to develop a third party component that interacts with the JVM. The component is developed as a dynamic library and is called a JVM TI agent.

7.6.2 Results

The articles presents an execution system, which cooperates with the JVM and incorporates a secure hypervisor, a JVM TI agent and an encryption tool. The encryption tool transforms regular Java program to encrypted Java programs which can be fed to the JVM just as regular programs. However, execution of such programs is possible only if the JVM TI agent is attached to the JVM.

The JVM TI agent intercepts attempts to execute encrypted code, constructs an execution environment and transfers control to the hypervisor. The hypervisor decrypts the code and begins interpretation. When the hypervisor reaches an uninterpretable instruction, it returns the control to the JVM TI agent and transfers the uninterpretable instruction in decrypted form. An instruction is considered uninterpretable if its interpretation requires cooperation with the JVM. The JVM TI agent interprets the instruction, and transfers control back to the hypervisor, if

the interpretation process shall continue.

Obviously, some information leaks during this process. The article estimates that 38% of the decrypted instructions are transferred to the JVM TI agent. This estimate is based on the statistical frequency of different instructions in an average Java program.

7.7 System for Executing Encrypted Native Programs

Kiperberg, M.; Leon, R.; Zaidenberg, N.J., "System for Executing Encrypted Native Programs" in *The Conferece*, Mon. Year

7.7.1 Research Problem

Digital content such as games, videos and the like may be susceptible to unlicensed usage, having 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 [Ore, VMP], which is vulnerable to hacking or user malicious activities [Rol09, Boh08]. There is therefore a need for a better technique for protecting sensitive software sections, such as licensing code.

7.7.2 Results

This paper presents a system that allows encrypting and executing native programs written for the x86 architecture.

The article presents an execution system, which consists of an encryption tool, a secure hypervisor and an authentication server. The encryption tool encrypts parts (functions) of a regular program, which can then be executed as a regular program, provided that the proposed system is installed on the target environment.

The decryption key is stored in the authentication server, which delivers it to the target machine upon a successful authentication. During the authentication protocol the authentication server verifies that the target machine is trustworthy, and that an authentic hypervisor is configured correctly. The decryption key, as well as any other sensitive information, is protected by the hypervisor. In addition to providing protection, the hypervisor decrypts and executes the encrypted functions.

YHTEENVETO (FINNISH SUMMARY)

Ohjelmakoodin takaisinmallintamisen estäminen on tärkeä keino ohjelmistojen suojaamiseen hyökkäyksiltä, algoritmien varastamiselta sekä laittomalta ohjelmiston käytöltä. Eräs yleisesti käytetty suojausmenetelmä on koodin tarkoituksellinen monimutkaistaminen, joka on kuitenkin osoittautunut tehottomaksi keinoksi. Ohjelmakoodin salaaminen sen sijaan on vahva takaisinmallintamisen estämismenetelmä, mikäli salausavainten hallinnointi pystytään toteuttamaan turvallisesti.

Tämä väitöskirja esittää menetelmän, joka mahdollistaa salatun ohjelmakoodin suorittamisen ja salausavainten hallinnan suojatussa ympäristössä. Menetelmä on turvallinen, koska salausavaimia ei missään vaiheessa tallenneta ohjelmakoodia suorittavalle tietokoneelle, vaan avaimet haetaan etäpalvelimelta tunnistautumisen jälkeen. Salausavaimet ja salauksesta avattu ohjelmakoodi pidetään jatkuvasti virtuaalikoneita ajavan ohjelman, hypervisorin, suojaamina. Menetelmä mahdollistaa sekä konekielisen, että Java ohjelmakoodin salaamisen ja suorittamisen.

Suojauksesta avattu ohjelmakoodi voidaan suojata suorituksen aikana haitalliselta ohjelmakoodilta joko estämällä muun ohjelmakoodin samanaikainen suorittaminen tai suojaamalla muistialue, jolla suojauksesta avatut komennot sijaitsevat. Konekielisistä ohjelmista poiketen Java-ohjelmat suoritetaan prosessorin sijasta Java virtuaalikoneessa (JVM). JVM tarvitsee salausavaimen salattujen ohjelmakoodin osien purkamiseen, eikä ole olemassa keinoa, jolla salausavaimet voitaisiin säilyttää turvallisesti JVM:n sisällä. Väitöskirjassa ehdotetaankin Java-tulkin implementointia ohuen hypervisorin ohjaaman suojatun ympäristön sisällä. Tulkkia ajetaan tavallisen JVM:n rinnalla, jolloin tulkki ja JVM yhdessä pystyvät suorittamaan salattuja Java-ohjelmia.

REFERENCES

[AMD10] AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2010.

[Bes80] R. M. Best. Preventing software piracy with crypto-microprocessors. In *Computer Society International Conference*, COMPCON'80, 1980.

[BH06] Walter Binder and Jarle Hulaas. Exact and portable profiling for the jvm using bytecode instruction counting. *Electronic Notes in Theoretical Computer Science*, 164(3):45–64, 2006.

[Boh08] Lutz Bohne. Pandora's Bochs: Automated Unpacking of Malware. 2008.

[Bri15] Brian. Nintendo starting to ban pirates from online services on 3ds. Technical report, Nintendo everything, 2015.

[BYDD+10] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *OSDI*, volume 10, pages 423–436, 2010.

[CFPS09] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the Difficulty of Software-based Attestation of Embedded Devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 400–409, New York, NY, USA, 2009. ACM.

[CMS07] Christian Collberg, Ginger Myles, and Michael Stepp. An Empirical Study of Java Bytecode Programs. *Softw. Pract. Exper.*, 37(6):581–641, May 2007.

[Cre81] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM J. Res. Dev.*, 25(5):483–490, September 1981.

[CSK10] Yosuke Chubachi, Takahiro Shinagawa, and Kazuhiko Kato. Hypervisor-based Prevention of Persistent Rootkits. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 214–220, New York, NY, USA, 2010. ACM.

[DKGC07] Daniel P Delorey, Charles D Knutson, and Christophe Giraud-Carrier. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *Second International Workshop on Public Data about Software Development (WoPDaSD'07)*, 2007.

[ELM+03] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, July 2003.

[Fer07] Peter Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, page 55, 2007.

[HAD10] Jason Howarth, Irfan Altas, and Barney Dalgarno. Information Flow Control Using the Java Virtual Machine Tool Interface (JVMTI). In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*, pages 689–695. IEEE, 2010.

[Int07] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3*, August 2007.

[Ion09] Daniel Ionescu. Microsoft bans up to one million users from xbox live. Technical report, PC World, 2009.

[KJ03] Rick Kennell and Leah H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 21–21, Berkeley, CA, USA, 2003. USENIX Association.

[Lon05] Frederick W Long. Software vulnerabilities in Java. 2005.

[Lue12] Mirko Luedde. Low impact debugging protocol, November 13 2012. US Patent 8,312,438.

[Ora] Oracle Corporation. *Java Virtual Machine Tool Interface*.

[Ore] Oreans. *Themida*.

[Pea02] Siani Pearson. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[Rol09] Rolf Rolles. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[RT07] Joanna Rutkowska and Alexander Tereshkin. IsGameOver(), anyone? In *Blackhat 2007*, 2007.

[RT08] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the xen hypervisor. *Black Hat USA*, 2008.

[SET+09] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko 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*, VEE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[SLP+06] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Scuba: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM Workshop on Wireless Security*, WiSe '06, pages 85–94, New York, NY, USA, 2006. ACM.

[SLS+05] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 1–16, New York, NY, USA, 2005. ACM.

[Son15] Sony. Information on banned accounts and consoles. Technical report, Sony consumer electronics, accessed on may 2015.

[SPvDK04] 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*, pages 272–282, May 2004.

[SWP08] Dries Schellekens, Brecht Wyseur, and Bart Preneel. Remote Attestation on Legacy Operating Systems with Trusted Platform Modules. *Sci. Comput. Program.*, 74(1-2):13–22, December 2008.

[SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[Tar10] Chris Tarnovsky. Semiconductor Security Awareness Today and yesterday. In *Blackhat*, 2010.

[Tar12] Chris Tarnovsky. Attacking TPM part two. In *Defcon*, 2012.

[VMP] VMProtect Software. *VMProtect*.

[Wik15] Wikipedia. An analysis of proposed attacks against genuinity tests. Technical report, accessed on May 2015.

[YHL+11] Qiang Yan, Jin Han, Yingjiu Li, Robert H. Deng, and Tieyan Li. A software-based root-of-trust primitive on multicore platforms. In

Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11, pages 334–343, New York, NY, USA, 2011. ACM.

[YWZC07] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 219–230, Washington, DC, USA, 2007. IEEE Computer Society.

ORIGINAL PAPERS

PI

REMOTE ATTESTATION OF SOFTWARE AND EXECUTION-ENVIRONMENT IN MODERN MACHINES

by

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

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

PII

TRUSTED COMPUTING AND DRM

by

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

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

PIII

EFFICIENT REMOTE AUTHENTICATION

by

Kiperberg, M.; Zaidenberg, N.J. 2013

The Journal of Information Warfare , vol.12, no.3

PIV

TRULY-PROTECT: AN EFFICIENT VM-BASED SOFTWARE PROTECTION

by

Averbuch, A.; Kiperberg, M.; Zaidenberg, N.J. 2013

Systems Journal, IEEE , vol.7, no.3, pp. 455-466

Truly-Protect: An Efficient VM-Based Software Protection

Amir Averbuch, Michael Kiperberg, and Nezer Jacob Zaidenberg, *Member, IEEE*

***Abstract*—We present Truly-Protect that is a software protection method. Previously published protection methods relied solely on obscurity. Rolles proposed a general approach for breaking systems that are based on obscurity. We show that, under certain assumptions, Truly-Protect is resistant not only to Rolles' attack but also to any other attacks that do not violate the assumptions. Truly-Protect is based on a virtual machine that enables us to execute encrypted programs. Truly-Protect can serve as a platform for preventing software piracy of obtaining unlicensed copies. Truly-Protect by itself is not a digital rights management system but can form a basis for such a system. We discuss several scenarios and implementations and validate the performance penalty of our protection. A preliminary version of this paper appeared in the 5th International Conference on Network and System Security (NSS2011). It was extended by expanding the system's description, adding more efficient parallel implementation, just-in-time decryption, and a comprehensive performance analysis. It also contains all the necessary proofs.**

***Index Terms*—Copy-protection, DRM, process virtual machine.**

I. Introduction

A RISING trend in the field of virtualization is the use of virtual machine (VM)-based digital rights and copy protection. The two goals of introducing VMs to digital rights protection are to encrypt and obfuscate the program. Forcing the hackers to migrate from a familiar x86 environment to an unfamiliar and obfuscated virtual environment is intended to pose a greater challenge in breaking the software copy protection.

A generic and semiautomatic method for breaking VM-based protection is proposed by Rolles [23]. It assumes that the VM is, broadly speaking, an infinite loop with a large switch statement called the op-code dispatcher. Each case in this switch statement is a handler of a particular op-code.

The first step a reverse engineer should take according to Rolles' method is to examine the VM and construct a translator. The translator is a software tool that maps the program instructions from the VM language to some other language chosen by the engineer, for example, x86 assembly. The VM may be stack based or register based. The reverse-engineer work is similar in both cases.

Rolles calls a language that the translators translate the code it reads into, i.e., an intermediate representation (IR). The first step is done only once for a particular VM-based protection, regardless of how many software systems are protected using the same VM. In the second step, the method extracts the VM op-code dispatcher and the obfuscated instructions from the executable code. The op-codes of these instructions, however, do not have to correspond to those of the translator: the op-codes for every program instance can be permuted differently. In the third step, the method examines the dispatcher of the VM and reveals the meaning of each op-code from the code executed by its handler. Finally, the obfuscated instructions are translated to IR. At this point, the program is not protected anymore since it can be executed without the VM. Rolles further applies a series of optimizations to achieve a program, which is close to the original one. Even by using Rolles' assumptions, we argue that a VM, which is unbreakable by the Rolles' method, can be constructed. In this paper, we will describe how to construct such a VM.

We do not try to obfuscate the VM. Its detailed description appears herein. We protect the VM by secretly holding the op-code dispatcher. By secretly we mean inside the CPU internal memory. Holding the op-code dispatcher in secret makes it impossible to perform the second step described by Rolles.

Moreover, we claim that the security of the system can be guaranteed under the following assumptions:

- The inner state of the CPU cannot be read by the user.
- The CPU has a sufficient amount of internal memory.

The former assumption simply states that the potential attacker cannot access the internal hardware of the CPU. In other words, we assume that the attacker cannot access the transistors that constitute the registers and the cache of the CPU through physical means. The second assumption, however, is more vague; hence, the properties of such an internal memory are discussed in Section VI.

The paper has the following structure: Section II provides an overview of related work. Section III outlines a step-by-step evolution of our system. Final details of the evolution are provided in Section IV. Section V describes the security of the proposed system. Section VI describes how to use different facilities of modern CPUs to implement the required internal memory. The performance is analyzed in Section VIII. Section IX provides an example of a C program and its corresponding encrypted bytecode.

Manuscript received October 1, 2011; revised March 15, 2012, July 20, 2012, and August 31, 2012; accepted September 1, 2012. Date of current version July 3, 2013. The work of N. J. Zaidenberg work was supported by the Graduate School in Computing and Mathematical Sciences (COMAS) and the Artturi and Ellen Nyyssösen Foundation.

A. Averbuch is with the School of Computer Science, Tel Aviv University, 39040 Tel Aviv, Israel (e-mail: amir@math.tau.ac.il).

M. Kiperberg and N. J. Zaidenberg are with the Department of Mathematical Information Technology, University of Jyväskylä, 40014 Jyväskylä, Finland (e-mail: mikiperb@student.jyu.fi; nezer.j.zaidenberg@jyu.fi).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSYST.2013.2260617

II. Related Work

A. Virtual Machines for Copy Protection

The two goals of introducing VM to trusted computing are to encrypt and to obfuscate the program. Forcing the hackers to migrate from a familiar x86 environment to an unfamiliar and obfuscated virtual environment is intended to pose a greater challenge in breaking the software copy protection.

However, not much has been published on the construction of VMs for digital rights protection as it would be counter productive for the obfuscation efforts that were the main reason for using VMs. Therefore, we often learn about VM protection from their breakers instead of their makers, for example, [5], [24], and [25].

Examples of using virtualization for copy protection include the hypervisor in Sony Play Station 3 [28] and XBox 360. The Cell OS Level 1 is a system VM that is not exposed to the user. The XBox 360 hypervisor also ensures that only signed code will be executed [7]. For PC software, Code Virtualizer [22] or VMProtect [32] are both software protection packages based on process VMs.

B. Hackers Usage of Virtualization

By running the protected software in a virtual environment, it became possible to disguise it as a different system. For example, running OS X on a standard hardware using a VM disguised as an Apple machine [10]. Protection methods against virtualization were researched in [15]. Inspecting and freezing the CPU and memory of a running system to obtain, for example, a copy of a copyrighted media, is another threat faced by media protectors such as NDS PC Show [21].

C. Execution Verification

Protecting and breaking software represent a long struggle between vendors and crackers that began even before VM protection gained popularity. Users demand a mechanism to ensure that the software they acquire is authentic. At the same time, software vendors require users to authenticate and ensure the validity of the software license for business reasons.

Execution verification and signatures is now a part of Apple Mach-O [1] object format and Microsoft Authenticode [19]. It also exists as an extension to Linux ELF [20].

The trusted components have been heavily researched in industry [31] and academia [29], among others, with mixed results. Both software rights protectors and hackers were able to report on a partial success.

From the hackers' camp, [30] is a fundamental paper dissecting all Microsoft's security mistakes in the first XBox generation.

III. System Evolution

Here, we describe the evolution of the proposed system in several phases which are fictional interim versions of the system. For each version, we describe the system and discuss its advantages, disadvantages, and fitness to today's world of hardware and software components.

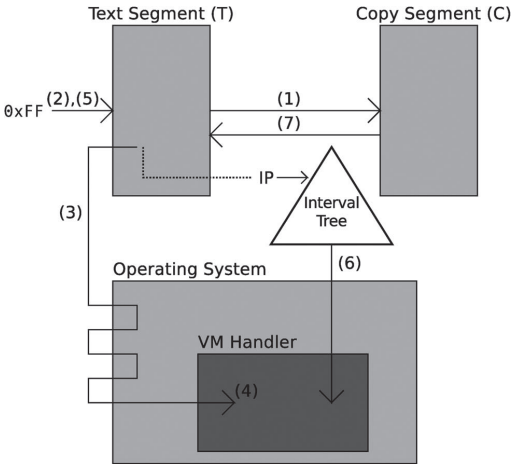

Fig. 1. Just-in-time decrypting.

We explore the means to improve the analyzed version and consider the implementation details worth mentioning, as well as any related work.

The first version describes the system broken by Rolles. The second version cannot be broken by Rolles but has much stronger assumptions. The rest of the evolution process consists of our engineering innovation. The sixth version is as secure as the second one but requires only the assumptions of the first version.

In the seventh version, we propose a completely different approach: a just-in-time decryption mechanism, which incurs only minor performance penalty (see Fig. 1).

The last section presents a parallelization scheme for the sixth version, which can theoretically improve its performance by utilizing an additional core present at a potential CPU. This idea was not implemented and thus described at the end of this section.

A. Dramatis Personae

The following are actors that participate in our system use cases:

1) *Hardware Vendor:* The Hardware Vendor is trustworthy and manufactor of the hardware used. The Hardware Vendor can identify components he manufactured. A real-world example is Sony as the Hardware Vendor of Play Station 3.
2) *Software Distributor:* Software Distributor distributes copy protected software. It is interested in providing conditional access to the software. In our case, the Software Distributor is interested in running the software on one End User CPU per license. A possible real-world example is VMProtect.
3) *"Game":* The software we wish to protect. It may be a computer game, a copyrighted video, or other piece of software.

4) *End User:* The End User may purchase legal copy of the Game from a Software Distributor. The End User may be interested in providing other users with illegal copies of the Game. The End User is not trustworthy. The goal of the system described herein is to prevent any of the End Users from obtaining even a single illegal copy of the Game.
5) *VM:* A software component developed and distributed by a Software Distributor.
6) *VM Interpreter:* A subcomponent of the VM that interprets the instructions given by the Game.
7) *VM Compiler:* A software component used by the Software Distributor to convert a Game code developed in a high-level programming language to instructions interpretable by the VM Interpreter.
8) *Malicious End User:* The Malicious End User would like to obtain illegitimate copy of the game. The VM Compiler, VM Interpreter, and VM are tools manufactured by the Software Distributor and Hardware Vendor that prevent the Malicious End User from achieving her goal: a single unlicensed copy. The Malicious End User may enlist one or more End User with legal copies of the game to achieve her goal.

B. Evolution

The building of the system will be described in steps.

1) System Version 1: The VM Interpreter represents a virtual unknown instruction set architecture (ISA) or a permutation of a known instruction set, such as MIPS, in our case. The VM Interpreter runs a loop:

Algorithm 1 System 1—VM Interpreter Run Loop

```
1: while VM is running do
2:    fetch next instruction
3:    choose the instruction handling routine
4:    execute the routine
5: end while
```

The VM Compiler reads a program in a high-level programming language and produces the output in the chosen ISA.

2) System Version 1—Discussion: Cryptographically speaking, this is an implementation of a replacement cipher on the instruction set. This method was described by Rolles [23] and used by VMProtect. Of course, the VM may include several other obfuscating subcomponents that may even provide greater challenge to a malicious user, but this is beyond our scope. The protection is provided by the VM complexity and by the user's lack of ability to understand it and, as stated previously, in additional obfuscations.

Rolles describes a semiautomatic way to translate a program from the unknown ISA to IR and later to the local machine ISA. Understanding how the VM works is based on understanding the interpreter. This problem is unavoidable. Even if the interpreter is implementing a secure cipher such as Advance Encryption Standard (AES), it will be unable to provide a tangible difference as the key to the cipher will also be stored in the interpreter in an unprotected form.

Therefore, it is vital to use a hardware component that the End User cannot reach to provide an unbreakable security.

3) System Version 2: The Hardware Vendor cooperates with the Software Distributor. He provides a CPU that holds a secret key known to the Software Distributor. Furthermore, the CPU implements an encryption and decryption algorithms.

The compiler needs to encrypt the program with the CPU's secret key. This version does not require a VM since the decryption takes place inside the CPU and its operation is similar to that of a standard computer.

4) System Version 2—Discussion: This version can implement any cipher, including AES, which is considered strong. This form of encryption was described by Best [3]. Some information about the program, such as memory access patterns, can still be obtained.

This method requires manufacturing processors with cryptographic functionality and secret keys for decrypting every fetched instruction. Such processors are not widely available today.

5) System Version 3: This system is based on system version 2, but the decryption algorithm is implemented in software. We alleviate the hardware requirements of system version 2. The CPU stores a secret key that is also known to the Software Distributor. The VM Compiler reads the Game in a high-level programming language and provides the Game in an encrypted form where every instruction is encrypted using the secret key. The VM knows how to decrypt the value stored in one register with a key stored in another register.

The VM Interpreter runs the following loop:

Algorithm 2 System 3—VM Interpreter Run Loop

```
1: while VM is running do
2:    fetch next instruction
3:    decrypt the instruction
4:    choose the instruction handling routine
5:    execute the routine
6: end while
```

6) System Version 3—Discussion: This version is as secure as system version 2, assuming that the VM internal state is stored at all times inside the CPU internal memory.

This method dramatically slows down the software. For example, decrypting one instruction using AES takes up to 112 CPU cycles.

7) System Version 4: System version 3 took a dramatic performance hit, which we now try to improve.

By combining versions 1 and 3, we implement a substitution cipher as in version 1. The cipher is polyalphabetic, and the special instructions embedded in the code define the permutation that will be used for the following instructions.

Similar to system version 3, we use the hardware for holding a secret key that is known also to the Software Distributor.

The VM Interpreter runs the following code:

Algorithm 3 System 4—VM Interpreter Run Loop

```
1: while VM is running do
2:    fetch next instruction
3:    decrypt the instruction
4:    if current instruction is not special then
5:       choose the instruction handling routine
6:       execute the routine
7:    else
8:       decrypt the instruction arguments using the secret key
9:       build a new instruction permutation
10:   end if
11: end while
```

8) System Version 4—Discussion: Section VIII defines a structure of the special instructions and a means to efficiently encode and reconstruct the permutations.

Dependent instructions should have the same arguments as justified by the following example, which is extracted from the Pi Calculator described in Section IX:

```
01: $bb0_1:
02: lw $2, 24($sp)
03: SWITCH (X)
04: lw $3, 28($sp)
05: subu $2, $2, $3
06: beq $2, $zero, $bb0_4
07: ...
08: $bb0_3:
09: $\ldots$
10: lw $3, 20($sp)
11: SWITCH (Y)
12: div $2, $3
13: mfhi $2
14: SWITCH (Z)
15: sw $2, 36($sp)
16: $bb0_4:
17: sw $zero, 32($sp)
18: lw $2, 28($sp)
```

This is a regular MIPS code augmented with three special instructions on lines 3, 11, and 14. The extraction consists of three basic blocks labeled bb0_1, bb0_3, and bb0_4. Note that we can arrive at the first line of bb0_4 (line 17) either from the conditional branch on line 6 or by falling through from bb0_3. In the first case, line 17 is encoded by X and in the second case, it is encoded by Z. The interpreter should be able to decode the instruction regardless of the control flow; thus, X should be equal to Z. In order to precisely characterize the dependence values between SWITCH instructions, we define the term "flow" and prove some facts about it.

Although a flow can be defined on any directed graph, one might want to imagine a control flow graph derived from some function. Then, every basic block of the graph corresponds to a vertex and an edge connecting x and y, suggesting that a jump from x to y might occur.

A flow comprises two partitions of all basic blocks. We call the partitions *left* and *right*. Every set of basic blocks from the left partition has a corresponding set of basic blocks from the right partition and vice versa. In other words, we can think of these partitions as that of a set of pairs. Every pair (A, B) has three characteristics: the control flow can jump from a basic block in A only to a basic block in B; the control flow can jump to a basic block in B only from a basic block in A; the sets A and B are minimal, in the sense that no basic blocks can be omitted from A and B.

The importance of these sets emerges from the following observation. In order to guarantee that the control flow arrives at a basic block in B with the same permutation, it is enough to make the last SWITCH instructions of basic blocks in A share the same argument. This is so, because we arrive at a basic block in B from some basic block in A. The formal proof follows.

Definition 1: Given a directed graph $G = (V, E)$ and a vertex $v \in V$. A flow is a pair (A_v, B_v) defined iteratively as follows:

- $v \in A_v$;
- If $x \in A_v$ then for every $(x, y) \in E$, $y \in B_v$;
- If $y \in B_v$ then for every $(x, y) \in E$, $x \in A_v$.

No other vertices appear in A or B.

A flow can be characterized in another way, which is less suitable for computation but simplifies the proofs. One can easily see that the two definitions are equivalent.

Definition 2: Given a directed graph $G = (V, E)$ and a vertex $v \in V$. A flow is a pair (A_v, B_v) defined as follows: $v \in A_v$ if there is a sequence $u = x_0, x_1, \ldots, x_k = v$ s.t. for every $1 \leq i \leq k$, there is $y_i \in V$ for which $(x_{i-1}, y_i), (x_i, y_i) \in E$. We call a sequence with this property a chain.

B_v is defined similarly.

We use the aforementioned definition to prove several lemmas on flows. We use them later to justify the characterization of dependent SWITCHes.

Since the definition is symmetric with respect to the chain direction, the following corollary holds.

Corollary 1: For every flow, $v \in A_u$ implies $u \in A_v$.

Lemma 1: If $v \in A_u$ then $A_v \subseteq A_u$.

Proof: A chain according to the definition is $u = x_0, x_1, \ldots, x_k = v$. Let $w \in A_v$, and let $v = x'_0, x'_1, \ldots, x'_{k'}$ be the chain that corresponds to w. The concatenation of these chains proves that $w \in A_u$. Therefore, $A_v \subseteq A_u$. ■

Lemma 2: If A_u and A_v are not disjoint then $A_u = A_v$.

Proof: A chain according to the definition is $u = x_0, x_1, \ldots, x_k = v$. Let $w \in A_u \cap A_v$. From the corollary, $u \in A_w$. The previous lemma implies that $A_u \subseteq A_w$ and $A_w \subseteq A_v$, thus $A_u \subseteq A_v$. The other direction can be proved in a similar manner. ■

Lemma 3: If A_u and A_v are not disjoint or if B_u and B_v are not disjoint, then $A_u = A_v$ and $B_u = B_v$.

Proof: We omit the proof since it is similar to the proof of the previous lemma. ■

Claim 1: Let $G = (V, E)$ be a control flow graph s.t. V is the set of basic blocks, and $(x, y) \in E$ if the control flow jumps from x to y. Two SWITCH instructions should share the same argument if and only if they are the last SWITCH instructions in

the basic blocks u and v s.t. $A_u = A_v$. We assume that every basic block contains a SWITCH instruction.

Proof: Consider the instruction γ. We need to prove that the interpreter arrives at γ with the same permutation, regardless of the execution path being taken.

If there is a SWITCH instruction α preceding γ in the same basic block, then every execution path passes through α in its way to γ; hence, the interpreter arrives at γ with the permutation set at α.

If there is no SWITCH instruction preceding γ in its basic block w, then consider two execution paths P and Q and let u and v be the basic blocks preceding w in P and Q, respectively. Denote by α the last SWITCH of u and by β the last SWITCH of v.

Clearly, $w \in B_u$ and $w \in B_v$, and thus by the last lemma, $A_u = A_v$. Therefore, α and β share the same argument, and on both paths, the interpreter arrives at γ with the same permutation. ■

The proposed system allows calling or jumping only to destinations known at compile-time, otherwise the dependency graph cannot be constructed reliably. Nevertheless, polymorphic behavior still can be realized. Consider a type hierarchy in which a function F is overridden. The destination address of a call to F cannot be determined at compile-time. Note however that such a call can be replaced by a switch statement, that dispatches to the correct function according to the source object type.

9) System Version 5: We rely on the previous version but give up on the assumption that the CPU is keeping a secret key that is known to the Software Distributor. Instead, we run a key-exchange algorithm [26].

Algorithm 4 Key Exchange in System 5

1: The Software Distributor publishes his public key.
2: The VM chooses a random number. The random number acts as the secret key. The random number is stored inside one of the CPU registers.
3: The VM encrypts it using a sequence of actions using the software distributor public key.
4: The VM sends the encrypted secret key to the Software Distributor.
5: The Software Distributor decrypts the value and gets the secret key.

10) System Version 5—Discussion: The method is secure if and only if we can guarantee that the secret key was randomized in a real (nonvirtual) environment where it is impossible to read CPU registers. Otherwise, it would be possible to run the program in a virtual environment where the CPU registers, and therefore, the secret key is accessible to the user. Another advantage of random keys is that different Games have different keys. Thus, breaking the protection of one Game does not compromise the security of others.

11) System Version 6: This version is built on top of the system described in the previous section. Initially, we run the verification methods described by Kennell and Jamieson [15]. Kennell and Jamieson propose a method of hardware and software verification that terminates with a shared "secret key" stored inside the CPU of the remote machine. The method is described in algorithm 5.

Algorithm 5 System 6—Genuine Hardware and Software Verification

1: The operating system (OS) on the remote machine sends a packet to the distributor containing information about its processor.
2: The distributor generates a test and sends a memory mapping for the test.
3: The remote machine initializes the virtual memory mapping and acknowledges the distributor.
4: The distributor sends the test (a code to be run) and public key for response encryption.
5: The remote machine loads the code and the key into memory and transfers control to the test code. When the code completes computing the checksum, it jumps to a (now verified) function in the OS that encrypts the checksum and a random quantity and sends them to the distributor.
6: The distributor verifies that the checksum is correct and the result was received within an allowable time, and if so, acknowledges the remote host of success.
7: The remote host generates a new session key. The session key acts as our shared secret key, concatenates it with the previous random value, encrypts them with the public key, and then sends the encrypted key to the distributor.

Using the algorithm of Kennell and Jamieson, we can guarantee the genuineness of the remote computer, i.e., the hardware is real and the software is not malicious.

The memory verified by algorithm 5 should reside in the internal memory of the CPU. This issue is discussed in greater detail in Section VI.

12) System Version 6—Discussion: System 6 alleviates the risk of running inside a VM that is found in system version 5.

13) System Version 7: Modern VMs, such as Java VM [18] and Common Language Runtime [4], employ just-in-time compilers to increase the performance of the program being executed. It is natural to extend this idea to the just-in-time decryption of encrypted programs. Instead of decrypting only a single instruction each time, we can decrypt an entire function. Clearly, decrypting such a large portion of the code is safe only if the CPU instruction cache is sufficiently large to hold it. When the execution leaves a decrypted function either by returning from it or by calling another function, the decrypted function is erased and the new function is decrypted. The execution continues. The benefit of this approach is obvious: every loop that appears in the function is decrypted only once, as opposed to being decrypted on every iteration by the interpreter. The relatively low cost of decryption allows us to use stronger and, thus, less efficient cryptographic functions, making this approach more resistant to cryptanalysis.

This approach uses the key-exchange protocol described in system version 6. We assume that there is a shared secret key between the Software Distributor and the End User. The

Software Distributor encrypts the binary program using the shared key and sends the encrypted program to the End User. The VM loads the program to the memory in the same fashion that the OS loads regular programs to the main memory. After the program is loaded and just before its execution begins, the VM performs the following steps:

1) Make a copy of the program's code in another location.
2) Overwrite the original program's code with some value for which the CPU throws an illegal op-code exception, e.g., 0xFF on x86.
3) Set a signal handler to catch the illegal op-code exception.

We call the memory location containing the illegal op-codes as the "text segment" or "T." The copy, which was made on the first step, is called the "copy segment" or "C." After performing these steps, the program execution begins and then immediately throws an illegal op-code exception. This, in turn, invokes the handler set on step 3.

This mechanism is similar to just-in-time compilation. The handler is responsible for:

1) realizing which function is absent;
2) constructing it.

The first step can be done by investigating the program stack. We begin by finding the first frame whose instruction pointer is inside T. The list of instruction pointers can be obtained through the "backtrace" library call. Next, we have to identify the function that contains this address. This can be done either by naively traversing the entire symbol table, giving us linear time complexity, or by noting that this problem can be solved by the "interval tree" data structure [6]. The interval tree provides a logarithmic complexity: each function is a memory interval that contains instructions. The instruction pointer is a point, and we want to find an interval that intersects with this point.

After finding the function F to be constructed in T, we can compute its location in C, copy F from C to T, and finally decrypt it in C.

Note that, in contrast to just-in-time compilers, we need to destroy the code of the previously decrypted function before handling the new function. The easiest way to do this is to write 0xFF over the entire text segment.

Algorithm 6 Just-In-Time Decryption

1: **while** Execution continues **do**
2: The program is copied from T to C.
3: T is filled with illegal instructions.
4: Illegal op-code exception is thrown and the OS starts handling this exception.
5: The execution is transferred to the VM handler.
6: T is filled with illegal instructions.
7: Intersection is found between the instruction pointer and an interval in the interval tree.
8: The corresponding function is copied from C to T and decrypted.
9: **end while**

14) System Version 7—Discussion: Nowadays, when CPUs are equipped with megabytes of cache, the risk of instruction eviction is low even if the entire functions of moderate size are decrypted at once. Moreover, we propose to hold in the cache several frequently used functions in a decrypted form. This way, as shown in Fig. 2, we improve the performance drastically. We did not explore in-depth the function erasure heuristic, i.e., which functions should be erased upon exit and which should remain. However, we believe that the naive approach described below will suffice, meaning it is sufficient to hold the most frequently used functions, such that the total size is limited by some fraction of the cache size. This can be easily implemented by allocating a counter for each function and counting the number of times the function was invoked.

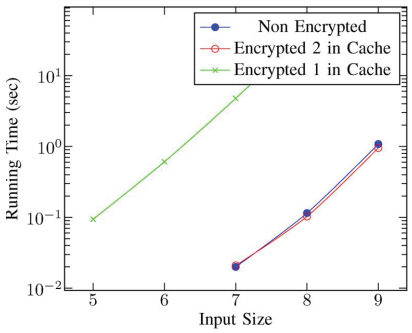

Fig. 2. Just-in-time decryption performance. Program running time (in seconds) as a function of input size. Note the logarithmic scale.

15) Parallel System—Future Work: Modern CPUs consist of multiple cores, and a cache is shared between these cores. This system is based on system version 6 that tries to increase its performance by utilizing the additional cores available on the CPU.

The key observation is that the decryption of the next instruction and the execution of the current instruction can be done in parallel on different cores. In this sense, we refer to the next instruction as the one that will be executed after the execution of the current instruction. Usually, the next instruction is the instruction that immediately follows the current instruction.

This rule, however, has several exceptions. If the current instruction is SWITCH, then the next instruction, decrypted by another thread, is decrypted with the wrong key. If the current instruction is a branch instruction, then the next instruction, decrypted by another thread, will not be used by the main thread. We call the instructions of these two types "special instructions." In all the other cases, the next instruction is being decrypted while the current instruction is executed.

These observations give us a distributed algorithm for the interpreter.

Algorithm 7 Core I Thread

1:**while** VM is running **do**
2: read instruction at $PC + 1$
3: decrypt the instruction
4: wait for Core II to execute the instruction at PC
5: erase (write zeros over) the decrypted instruction
6: **end while**

Algorithm 8 Core II Thread

```
1: while VM is running do
2:   if previous instruction was special then
3:     decrypt instruction at PC
4:   end if
5:   fetch next instruction at PC
6:   choose instruction handler routine
7:   execute instruction using handler routine
8:   if previous instruction was special then
9:     erase (write zeros over) the decrypted instruction
10:  end if
11:  wait for Core I to decrypt the instruction at PC + 1
12: end while
```

16) System Version 7—Discussion: We did not implement the proposed system, and it is a work in progress. Clearly, it can substantially increase the system performance. Note that we benefit here from a polyalphabetic cipher, since it is practically impossible to use a block cipher, such as AES, in this context. Block ciphers operate on large portions of plaintext or ciphertext; hence, they may require the decryption of many instructions at once. After branching to a new location, we will have to find the portion of a program that was encrypted with the current instruction and decrypt all of them. Obviously, this is far from being optimal.

IV. Final Details

A. Scenario

Here, we provide a scenario that involves all the dramatis personae. We have the following participants: Victor—a Hardware Vendor, Dan—a Software Distributor, Patrick—a programmer developing PatGame, and Uma—an End User.

Uma purchased a computer system supplied by Victor with Dan's VM preinstalled as part of the OS. Patrick, who wants to distribute his Game, sends it to Dan. Dan updates his online store to include PatGame as a new Game.

Uma, who wants to play PatGame, sends a request for PatGame to Dan via his online store. Dan authenticates Uma's computer system, possibly in cooperation with Victor, as described in system version 6. After the authentication is successfully completed, Uma's VM generates a random secret key R, encrypts it with Dan's public key D, and sends it to Dan. Dan decrypts the message obtaining R. This process was described in version 5. As described in version 4, Dan compiles the PatGame with the key R and sends it to Uma. Uma's VM executes the PatGame decrypting the arguments of special instructions with R.

A problem arises when Uma's computer is rebooted, since the key R is stored in a volatile memory. Storing it outside the CPU will compromise its secrecy, and thus the security of the whole system. We propose to store the key R on Dan's side.

Suppose Uma wants to play an instance of PatGame already residing on her computer. Uma's VM generates a random secret key Q, encrypts it with Dan's public key D, and sends it to Dan. Dan authenticates Uma's computer. After the authentication successfully completes, Dan decrypts the message obtaining Q. Dan encrypts the stored key R with the key Q, using AES for example, and sends it back to Uma. Uma decrypts the received message obtaining R, which is the program's decryption key. Thus, the encrypted program does not have to be sent after every reboot of Uma's computer.

B. Compilation

Since the innovation of this paper is mainly the fourth version of the system, we provide here a more detailed explanation of the compilation process. See Section IX for an example program passing through all the compilation phases.

The compiler reads a program written in a high-level programming language. It compiles it as usual up to the phase of machine code emission. The compiler then inserts new special instructions, which we call SWITCH, at random with probability p before any of the initial instructions. The argument of the SWITCH instruction determines the permutation applied on the following code up to the next SWITCH instruction. Afterwards, the compiler calculates the dependence values between the inserted SWITCH instructions. The arguments of the SWITCH instructions are set randomly but with respect to the dependence values.

The compiler permutes the instructions following SWITCH according to its argument. In the final pass, we encrypt the arguments of all SWITCHes by AES with the key R.

C. Permutation

In order to explain how the instructions are permuted, we should describe first the structure of the MIPS ISA we use. Every instruction starts with a 6-bit op-code that includes up to three 5-bit registers and, possibly, a 16-bit immediate value. The argument of the SWITCH instruction defines some permutation σ over 2^6 numbers and another permutation τ over 2^5 numbers. σ is used to permute the op-code, and τ is used to permute the registers. Immediate values can be encoded either by computing them, as described by Rolles [23], or by encrypting them using AES.

V. Security

The method described by Rolles requires a complete knowledge of the VM's interpreter dispatch mechanism. This knowledge is essential for implementing a mapping from bytecode to IR. In the described system, a secret key, which is part of the dispatch mechanism, is hidden from an adversary. Without the secret key, the permutations are secretly constructed. Without the permutations, the mapping from bytecode to IR cannot be reproduced.

The described compiler realizes an autokey substitution cipher. This class of ciphers is, on one hand, more secure than the substitution cipher used by VMProtect, and, on the other hand, does not suffer from the performance penalties typical to the more secure AES algorithm.

As discussed by Goldreich [9], some information can be gathered from memory accessed during program execution.

The author proposes a way to hide the access patterns, thus not allowing an adversary to learn anything from the execution.

In an effort to continue improving the system performance, we have considered using an efficient low-level VM [17]. Unfortunately, modern VMs with efficient just-in-time compilers are unsuitable for software protection. Once the VM loads the program, it allocates data structures representing the program, which are stored unencrypted in memory. Since this memory can be evicted from cache at any time, these data structures become a security threat in a software protection system.

VI. Assumptions in Modern CPUs

We posed two assumptions on the CPU that guarantee the security of the entire system. This section discusses the application of the system to the real-world CPUs. In other words, we show how to use the facilities of modern CPUs to imply the assumptions.

Let us first recall the following assumptions:

- The inner state of the CPU cannot be read by the user.
- The CPU has a sufficient amount of internal memory.

As to the later assumption, we should first clarify the purpose of the internal memory. In essence, this memory stores three kinds of data. The first one is the shared secret key. The second is the state of the VM, specifically the current permutation and the decrypted instruction. The third kind of data is some parts of the kernel code and the VM code. The reason behind the last kind is less obvious; therefore, consider the following attack.

An attacker lets the algorithm of Kennell and Jamieson to complete successfully on a standard machine equipped with a special memory. This memory can be modified externally and not by the CPU. Note that no assumption prohibits the attacker to do so. Just after the completion of the verification algorithm, the attacker changes the memory containing the code of the VM to print every decrypted instruction. Clearly, this breaks the security of the proposed system. Observe that the problem is essentially the volatility of critical parts of the kernel code and the VM code. To overcome this problem, we have to disallow modification of the verified memory. Since the memory residing inside the CPU cannot be modified by the attacker, we can assume it to remain unmodified.

Note that the first and second kinds, which hold the shared secret key and the VM's state, should be readable and writeable, whereas the third kind, which holds the critical code, should be only readable.

On Intel CPUs, we propose to use registers as a storage for the shared secret key and the VM's state and to use the internal cache as a storage for the critical code.

To protect the key and the state, we must store them in registers that can be accessed only in the kernel mode. On Intel CPUs, only the special purpose segment registers cannot be accessed in the user mode. Since these registers are special, we cannot use them for our purposes. However, on 64-bit CPUs running a 32-bit OS, only half of the bits in these registers are used to provide the special behavior. The other half can be used to store our data.

The caching policy in Intel CPUs can be turned on and off. The interesting thing is that, after turning it off, the data are not erased from the cache. Subsequent reads of these data return what is stored in the cache even if the memory has been modified. We use this property of the cache to extend the algorithm of Kennell and Jamieson not only to validate the code of the kernel and the VM before the key generation but also to guarantee that the validated code will never change. Two steps should be performed just after installing the virtual memory mapping received from the distributor in the verification algorithm: loading the critical part of the kernel and the VM program code (i.e., the TEXT segments) into the cache and turning the cache off. This behavior of Intel CPUs is documented in [14].

The first assumption disallows the user to read the aforementioned internal state of the CPU physically, i.e., by opening its case and plugging wires into the CPU's internal components. Other means of accessing the internal state are controlled by the kernel and hence are guaranteed to be blocked.

VII. Counterattacks

Here, we present possible counterattacks on the presented system. For each counterattack, we propose a way to accommodate it.

The success of our approach depends on the secrecy of the *secret key* and the *VM's internal state*. We claimed that it is possible to guarantee that both the key and the state never leave the CPU. This is indeed true during a regular operation of the CPU, i.e., this information cannot be accessed by a malicious software and it never appears on the system bus. However, some Intel CPUs are equipped with Test Access Port (TAP) [12], which can be used to fetch the values of all registers. The work through TAP is done solely by hardware means; hence, a software has no ability to prevent it. We leave it an open question if TAP can be detected using side-effect-based detection.

To the best of our knowledge, there is no mechanism similar to TAP on AMD CPUs. If this is so, AMD CPUs can be used to realize our system. Moreover, TAP is not an essential part of the CPU, it was initially designed to test printed circuit boards. Hence, we are convinced that removing TAP from Intel CPUs should not be difficult.

An essential building block of our system is a protocol that verifies the software and hardware genuineness. This paper is based on a protocol described in [15]. Any attack on this protocol invalidates our approach too. However, in this paper, we assume that such a protocol exists. The assumption is reasonable, since such protocols can be developed, attacked, and fixed in parallel to our work and independently of it. For completeness, we note that, soon after the publication of [15], an attack [27] was proposed, which was followed by a clarification of the original authors [16]. Therefore, the protocol can be considered intact. In case an attack on [15] is found, some of the recently published tests [8] can be used as an alternative.

Another essential building block of our system is the cryptographic algorithm we use to encrypt and decrypt permutations. Currently, we use the AES algorithm. To the best of our knowledge, there is no information theoretic quantization of

the amount of security achieved by AES, or any other block cipher. Therefore, we cannot guarantee that AES and, as a consequence, our system cannot be attacked. The National Security Agency stated that the strength of AES is sufficient to protect classified information [11]. We rely on this statement for the time being. Anyhow, AES can be replaced by any other block cipher, without any modifications of our system.

VIII. Performance

Here, we analyze in detail the performance of the proposed cipher. Throughout this section, we compare our cipher to AES. This is due to recent advances in CPUs that make AES to be the most appropriate cipher for program encryption [13]. We provide both theoretical and empirical observations proving our algorithm's supremacy.

We denote the number of cycles needed to decrypt one word of length w using AES by α.

The last subsection compares system version 7 to a regular unprotected execution.

A. Version 3 Performance

Consider version 3 of the proposed system and assume it uses AES as a cipher. Every instruction occupies exactly one word, such that n instructions can be decrypted in $n\alpha$ cycles.

B. Switch Instructions

As previously described, the SWITCH instruction is responsible for choosing the current permutation σ. This permutation is then used to decrypt the op-codes of the following instructions.

Some details were previously omitted, since they affect only the system's performance but do not affect its security or overall design.

- How does the argument of a switch instruction encode the permutation?
- Where is the permutation stored inside the processor?

Before answering these questions, we introduce two definitions.

Definition 3: Given an encrypted program, we denote the set of all instructions encrypted with σ by I_σ and call it color-block σ.

Definition 4: Given a set I of instructions, we denote by $D(I)$ the set of different instructions (those having different op-codes) in I.

The key observation is that it is enough to define how σ acts on the op-codes of $D(I_\sigma)$, which are instructions that occur in the color-block σ. Likewise, we noticed that some instructions are common to many color-blocks, while others are rare.

Denote by $F = \{f_1, f_2, \ldots, f_\ell\}$ the set of the ℓ most frequent instructions. Let $R(I)$ be the set of rare instructions in I, i.e., $R(I) = D(I) - F$. The argument of SWITCH preceding a color-block σ has the following structure (and is encrypted by AES, as described in version 5):

$$\sigma(f_1), \sigma(f_2), \ldots, \sigma(f_\ell),$$
$$r_1, \sigma(r_1), r_2, \sigma(r_2), \ldots, r_k, \sigma(r_k)$$

TABLE I
Correlation Between ℓ, ϕ, and q. Here, $p = 0.2$

ℓ	0	1	2	3	4	5	6
ϕ	58	39	42	41	58	46	48
$\frac{q}{n}$	100%	70%	50%	34%	34%	26%	21%

where $R(I_\sigma) = \{r_1, r_2, \ldots, r_k\}$. If σ acts on m-bit-long op-codes, then the length of the encoding of σ is $\phi = (\ell + 2k)m$ bits. Thus, its decryption takes $((\ell + 2k)m/w)\alpha$ cycles.

C. Version 4 Performance

Consider a sequence of instructions between two SWITCHes in the program's control flow. Suppose these instructions belong to I_σ and the sequence is of length n. The VM Interpreter starts the execution of this sequence by constructing the permutation σ. Next, the VM Interpreter goes over all the n instructions, decrypts them according to σ, and executes them, as described in version 5.

The VM Interpreter stores $\sigma(f_1), \sigma(f_2), \ldots, \sigma(f_\ell)$ in the CPU registers and the rest of σ in the internal memory. This allows decryption of frequent instructions in one cycle. Decryption of rare instructions takes $\beta + 1$ cycles, where β is the internal memory latency in cycles. Denote by q the number of rare instructions in the sequence.

We are now ready to compute the number of cycles needed to decrypt the sequence

$$\sigma \text{ construction} : \frac{(\ell + 2k)m}{w}\alpha +$$
$$\text{frequent instructions} : (n - q) +$$
$$\text{rare instructions} : q(\beta + 1).$$

D. Comparison

On MIPS op-codes are $m = 6$ bits long and $w = 32$. The best available results for Intel newest CPUs argue that $\alpha = 14$ [33]. Typical CPUs' Level-1 cache latency is $\beta = 3$ cycles. The correlation between ℓ, ϕ, and q is depicted in Table I.

We have noticed that most of the cycles are spent constructing permutations, and this is done every time SWITCH is encountered. The amount of SWITCHes, and thus the time spent constructing permutations, varies with the probability p of SWITCH insertion. The security, however, varies as well. Fig. 3 compares program decryption time (in cycles) using AES and our solution with different values of p.

Note that on CPUs not equipped with AES/GCM [33], such as Pentium 4, $\alpha \geqslant 112$. In this case, our solution is even more beneficial. Fig. 4 makes the comparison.

E. Version 7 Performance

The performance is affected mainly by the amount of function calls, since each call to a new function requires the function decryption. This performance penalty is reduced by allowing several functions to be stored in a decrypted form simultaneously.

Fig. 2 compares the running times of the same program in three configurations and with inputs of different sizes. The

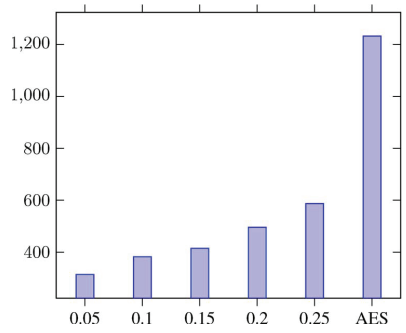

Fig. 3. Program decryption time (in cycles) using AES and our cipher with different values of p. $\ell = 4$. $\alpha = 14$.

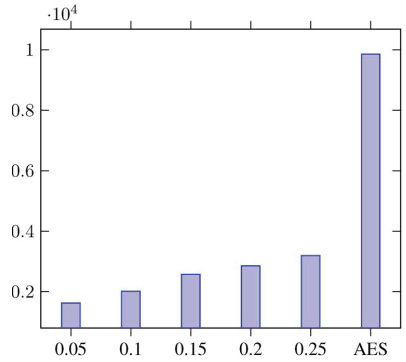

Fig. 4. Program decryption time (in cycles) using AES and our cipher with different values of p. $\ell = 4$. $\alpha = 112$.

program inverses the given matrix using Cramer's rule. The program consists of three functions computing determinant, minor, and finally inversion of a square matrix. The determinant is recursively computed, reducing the matrix order on each step by extracting its minor. We run this program on matrices of different sizes.

The three configurations of the program include:

1) nonencrypted configuration—just a regular execution;
2) encrypted configuration allowing two functions to reside in the cache simultaneously;
3) encrypted configuration allowing a single function to reside in the cache simultaneously.

F. Comparison to Non-Protected System

Fig. 5 demonstrates the protected program execution time when compared to compiled and interpreted programs on without protection. For comparison purposes, we use both instruction switching and AES encryption for protection.

IX. Example

We provide a detailed example of a program passing through all the compilation phases. The original program is written

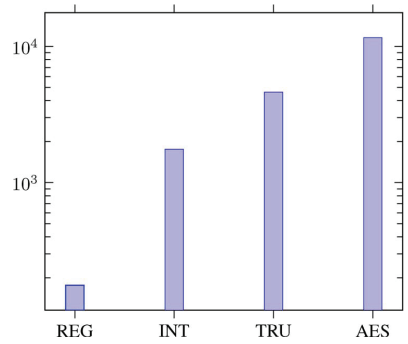

Fig. 5. Program decryption time (in cycles) using (left to right) a regular execution, interpretation, AES, and our cipher. $\ell = 4$. $\alpha = 112$. $p = 0.2$. Note the logarithmic scale.

in C. It computes the sum of the first 800 digits of π.

```
int main(){
  int a = 10000, b, c = 2800, d, e,
  f[2801], g, s;
  for (;b-c;) f[b++] = a/5;
  for (;d = 0, g = c*2; c -= 14, e = d%a) {
    for (b = c;
      d += f[b]*a, f[b] = d%--g, d /= g--,
      --b;
          d *= b);
        s += e + d/a;
  }
  return s;
}
```

The corresponding output of the compiler is a combination of MIPS assembly and MIPS machine instructions. Block labels are emphasized.

Instructions of the form

```
addiu $zero, $zero, ...
```

are an implementation of the SWITCH instruction. Colors correspond to the permutation that should be applied to the instructions.

For example, consider the instruction at 000c (see Fig. 6). It sets the current permutation to 5 (corresponding to gray color). As a result, all following instructions (up to the next SWITCH) are colored gray. Note that the instruction at 000c is not colored gray, since it should be encoded by the previous permutation (pink, corresponding to number 4 and set at 0005).

After the instructions are permuted according to their colors, the final phase takes place: the compiler encrypts special instructions' arguments using AES with the secret key.

X. Summary

We have discussed several steps toward software protection. Our system has been designed only to mask the code of the

```
$main (1)
0001 24000002 addiu $zero, $zero, 2

0002 24000003 addiu $zero, $zero, 3

0003 27bdd400 addiu $sp, $sp, -11264
0004 24022710 addiu $2, $zero, 10000
0005 24000004 addiu $zero, $zero, 4

0006 afa00010 sw $zero, 16($sp)
0007 24030af0 addiu $3, $zero, 2800
0008 afa20014 sw $2, 20($sp)
0009 afa3001c sw $3, 28($sp)
$bb0_1 (4)
000b 8fa20018 lw $2, 24($sp)
000c 24000005 addiu $zero, $zero, 5

000d 8fa3001c lw $3, 28($sp)
000e 00431023 subu $2, $2, $3
000f 10020000 beq $2, $zero, $bb0_4

0011 3c026666 lui $2, 26214
0012 34426667 ori $2, $2, 26215
0013 8fa30014 lw $3, 20($sp)
0014 8fa40018 lw $4, 24($sp)
0015 00620018 mult $3, $2
0016 00001010 mfhi $2
0017 00400803 sra $3, $2, 1
0018 24000006 addiu $zero, $zero, 6

0019 0040f801 srl $2, $2, 31
001a 27a50030 addiu $5, $sp, 48
001b 00801000 sll $6, $4, 2
001c 24840001 addiu $4, $4, 1
001d 24000004 addiu $zero, $zero, 4

001e 00621021 addu $2, $3, $2
001f 00a61821 addu $3, $5, $6
0020 afa40018 sw $4, 24($sp)
0021 ac620000 sw $2, 0($3)
0022 0800000a j $bb0_1
$bb0_3 (7)
0024 8fa20020 lw $2, 32($sp)
0025 24000008 addiu $zero, $zero, 8

0026 8fa30014 lw $3, 20($sp)
0027 0043001a div $2, $3
0028 00001012 mflo $2
0029 8fa30024 lw $3, 36($sp)
002a 8fa42bf8 lw $4, 11256($sp)
002b 00621021 addu $2, $3, $2
002c 00821021 addu $2, $4, $2
002d afa22bf8 sw $2, 11256($sp)
002e 8fa2001c lw $2, 28($sp)
002f 2442fff2 addiu $2, $2, -14
0030 afa2001c sw $2, 28($sp)
0031 8fa20020 lw $2, 32($sp)
0032 8fa30014 lw $3, 20($sp)
0033 24000009 addiu $zero, $zero, 9

0034 0043001a div $2, $3
0035 00001010 mfhi $2
0036 24000005 addiu $zero, $zero, 5

0037 afa20024 sw $2, 36($sp)
$bb0_4 (5)
0039 afa00020 sw $zero, 32($sp)
003a 8fa2001c lw $2, 28($sp)
003b 00400800 sll $2, $2, 1
003c afa22bf4 sw $2, 11252($sp)
003d 10020000 beq $2, $zero, $bb0_8

003f 8fa2001c lw $2, 28($sp)
0040 afa20018 sw $2, 24($sp)
$bb0_6 (5)
0042 8fa20018 lw $2, 24($sp)
0043 27a30030 addiu $3, $sp, 48
0044 00401000 sll $2, $2, 2
0045 00621021 addu $2, $3, $2
0046 2400000a addiu $zero, $zero, 10

0047 8c420000 lw $2, 0($2)
0048 8fa40014 lw $4, 20($sp)
0049 8fa50020 lw $5, 32($sp)
004a 00440018 mult $2, $4
004b 2400000b addiu $zero, $zero, 11

004c 00001012 mflo $2
004d 00a21021 addu $2, $5, $2
004e afa20020 sw $2, 32($sp)
004f 8fa42bf4 lw $4, 11252($sp)
0050 2484ffff addiu $4, $4, -1
0051 afa42bf4 sw $4, 11252($sp)
0052 8fa50018 lw $5, 24($sp)
0053 00a01000 sll $5, $5, 2
0054 0044001a div $2, $4
0055 00001010 mfhi $2
0056 00651821 addu $3, $3, $5
0057 ac620000 sw $2, 0($3)
0058 8fa22bf4 lw $2, 11252($sp)
0059 2400000c addiu $zero, $zero, 12

005a 2443ffff addiu $3, $2, -1
005b afa32bf4 sw $3, 11252($sp)
005c 8fa30020 lw $3, 32($sp)
005d 0062001a div $3, $2
005e 00001012 mflo $2
005f afa20020 sw $2, 32($sp)
0060 24000007 addiu $zero, $zero, 7

0061 8fa20018 lw $2, 24($sp)
0062 2442ffff addiu $2, $2, -1
0063 afa20018 sw $2, 24($sp)
0064 10020000 beq $2, $zero, $bb0_3

0066 8fa20018 lw $2, 24($sp)
0067 2400000d addiu $zero, $zero, 13

0068 8fa30020 lw $3, 32($sp)
0069 00620018 mult $3, $2
006a 00001012 mflo $2
006b 24000005 addiu $zero, $zero, 5

006c afa20020 sw $2, 32($sp)
006d 08000041 j $bb0_6
$bb0_8 (5)
006f 8fa22bf8 lw $2, 11256($sp)
0070 27bd2c00 addiu $sp, $sp, 11264
0071 03e00008 jr $ra
```

Fig. 6. MIPS machine instructions.

software from a malicious user. The system can be used to prevent the user from either reverse engineering the Game or to create an effective key validation mechanism. It does not prevent access to the Game data, which may be stored unprotected in memory. Even if the entire Game is encoded with this system, a player may still hack the Game data to affect his high score, credits, or "lives." A different encryption mechanism, which can be protected by Truly-Protect, can be added to prevent it. For similar reasons, the system cannot be used to prevent copying of copyrighted content, such as movies

and audio, unless the content is also encrypted. Truly-Protect can be used to mask the decoding and decryption process.

ACKNOWLEDGMENT

The authors would like to thank D. Sotnikov for his brilliant ideas and his valuable comments on both forms and content of this paper.

REFERENCES

[1] Apple Ltd. Mac OS X ABI Mach-O File Format Reference. [Online]. Available: http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html
[2] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, "An efficient VMBased software protection," in *Proc. NSS*, 2011, pp. 121–128.
[3] R. M. Best, "Preventing software piracy with crypto-microprocessors," in *Proc. IEEE Spring COMPCON*, Feb. 1980, pp. 466–469.
[4] D. Box, *Essential.NET, Volume 1: The Common Langugage Runtime*. Reading, MA, USA: Addison-Wesley, 2002.
[5] Bushing, Marcan, and Sven, "Console hacking 2010 PS3 epic fail," in *Proc. CCC 2010: We Come in Peace*, 2010.
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Interval tree," in *Introduction to Algorithms*, 2nd ed. Cambridge, MA, USA: MIT Press, 2001, ch. 14.3, pp. 311–317.
[7] F. Domka and M. Steil, "Why silicon-based security is still that hard: Deconstructing Xbox 360 security," in *Proc. CCC*, 2007.
[8] P. Ferrie, "Attacks on virtual machine emulators," Symantec Adv. Threat Res., Mountain View, CA, USA, Tech. Rep., 2006.
[9] O. Goldreich, "Toward a theory of software protection," in *Proc. Adv. CRYPTO*, 1987, pp. 426–439.
[10] A. Graf, "Mac OS X in KVM," in *Proc. KVM Forum*, 2008.
[11] L. Hathaway, "National policy on the use of the advanced encryption standard (AES) to protect national security systems and national security information," NIST, Gaithersburg, MD, USA, Tech. Rep., 2003.
[12] *Intel Itanium 2 Processor, Hardware Developer's Manual*, Intel Corp., Santa Clara, CA, USA, 2001.
[13] *Breakthrough AES Performance With Intel AES New Instructions*, Intel Corp., Santa Clara, CA, USA, 2010.
[14] *Intel 64 and IA-32 Architectures Developer's Manual*, Intel Corp., Santa Clara, CA, USA, 2012.
[15] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *Proc. 12th USENIX Secur. Symp.*, 2003, p. 21.
[16] R. Kennell and L. H. Jamieson, "An analysis of proposed attacks against genuinity tests," CERIAS, Purdue Univ., West Lafayette, IN, USA, Tech. Rep. 2004-27, 2004.
[17] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. CGO: Feedback-Directed Runtime Optim.*, 2004, p. 75.
[18] T. Lindoholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1999.
[19] Microsoft Cooperation, Windows Authenticode Portable Executable Signature Form. [Online]. Available: http://msdn.microsoft.com/en-us/windows/hardware/gg463180.aspx
[20] millerm. elfsign. [Online]. Available: http://freshmeat.net/projects/elfsign/
[21] NDS. PC Show. [Online]. Available: http://www.nds.com/solutions/pc_show.php
[22] Oreans Technologies. Code Virtualizer. [Online]. Available: http://www.oreans.com/products.php
[23] R. Rolles, "Unpacking virtualization obfuscators," in *Proc. 4th USENIX WOOT*, 2009, p. 1.
[24] R. E. Rolles, Unpacking VMProtect. [Online]. Available: http://www.openrce.org/blog/view/1238/
[25] Scherzo, Inside Code Virtualizer. [Online]. Available: http://rapidshare.com/files/16968098/Inside_Code_Virtualizer.rar
[26] B. Schneier, "Key-exchange algorithms," in *Applied Cryptography*, 2nd ed. Hoboken, NJ, USA: Wiley, 1996, ch. 22.
[27] U. Shankar, M. Chew, and J. D. Tygar, "Side effects are not sufficient to authenticate software," in *Proc. 13th USENIX Secur. Symp.*, Aug. 2004, p. 7.
[28] SONY Consumer Electronics, Playstayion 3. [Online]. Available: http://us.playstation.com/ps3/
[29] M. Srivatsa, S. Balfe, K. G. Paterson, and P. Rohatgi, "Trust management for secure information flows," in *Proc. 15th ACM Conf. Comput. Commun. Security*, Oct. 2008, pp. 175–188.
[30] M. Steil, "17 mistakes Microsoft made in the XBox security system," presented at the 22nd Chaos Communication Congr., Berlin, Germany, 2005.
[31] Trusted Computing Group, TPM Main Specification. [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpmmainspecification
[32] VMPSoft. VMProtect. [Online]. Available: http://www.vmprotect.ru
[33] Wikipedia, the Free Encyclopedia. AES Instruction Set.

Amir Averbuch was born in Tel Aviv, Israel. He received the B.Sc. and M.Sc. degrees in mathematics from the Hebrew University of Jerusalem, Jerusalem, Israel, in 1971 and 1975, respectively, and the Ph.D. degree in computer science from Columbia University, New York, NY, USA, in 1983.

During 1966–1970 and 1973–1976, he served with the Israeli Defense Forces. During 1976–1986, he was a Research Staff Member with the Department of Computer Science, IBM Thomas J. Watson Research Center, Yorktown Heights, NY. In 1987, he joined the School of Computer Science, Tel Aviv University, Tel Aviv, where he is now a Professor of computer science. His research interests include applied harmonic analysis, wavelets, signal/image processing, security, numerical computation, and scientific computing (fast algorithms).

Michael Kiperberg was born in Ukraine, in 1987, and migrated to Ashkelon, Israel. He received B.Sc. degree (*cum laude*) in computer science and the M.Sc degree (*cum laude*) from Tel Aviv University, Tel Aviv, Israel. in 2009 and 2012, respectively. He is currently working toward the Ph.D. degree in the Department of Mathematical Information Technology, University of Jyväskylä, Jyväskylä, Finland, under the supervision of N. Zaidenburg.

In 2009, he joined the Israeli Defense Forces. He is currently serving as an Academic Officer with the Israeli Air Force.

Nezer Jacob Zaidenberg (M'11) was born in Tel Aviv, Israel, in 1979. He received B.Sc. degree in computer science and statistics and operations research, the M.Sc. degree in operations research, and the MBA degree from Tel Aviv University, Tel Aviv, in 1999, 2001, and 2006, respectively, and the Ph.D. degree from the University of Jyväskylä, Jyväskylä, Finland, in 2012.

He is currently a Postdoctoral Researcher with the University of Jyväskylä.

PV

AN EFFICIENT VM-BASED SOFTWARE PROTECTION

by

Averbuch, A.; Kiperberg, M.; Zaidenberg, N.J. 2011

Network and System Security (NSS), 2011 5th International Conference, pp. 121-128

An Efficient VM–Based Software Protection

Amir Averbuch
Tel Aviv University
P.O.Box 39040
Ramat Aviv, Israel
Email: amir@math.tau.ac.il

Michael Kiperberg
Tel Aviv University
P.O.Box 39040
Ramat Aviv, Israel
Email: kiperber@post.tau.ac.il

Nezer Jacob Zaidenberg
University of Jyväskylä
P.O.Box 35, FI-40014
Jyväskylä, Finland
Email: nezer.j.zaidenberg@jyu.fi

***Abstract*—This paper presents Truly–protect, a system, incorporating a virtual machine, that enables execution of encrypted programs. Our intention is to form a framework for a conditional access/digital rights management system.**

We avoid relying on obscurity and rely only on assumptions about the system itself and on cryptographic measures to develop VM–based conditional access/trusted computing environment.

Rolles in [18], proposes a general way of breaking systems of type described herein. We claim that Rolles' method fails to defeat our system.

I. Introduction

In recent years, there has been a steady growth in the field of virtualization. "Virtual machines" are separated into two major categories: process virtual machines and system virtual machines.

Process virtual machine, such as JVM[13] or CLR[5], are controlled environments for running processes.

System virtual machines, such as VMWare[31], Xen[2] or KVM[12], [3], are virtual environments for running complete operating systems. These environments can run directly above the hardware (in which case they are called "Type-1 hypervisor") such as VMWare ESX server, or as a process running on host OS (in which case they are called "Type-2 hypervisor") such as QEMU.

In industry, virtual machines (VM) of both types are used to provide multiple cross cutting aspects such as replication[7], sandboxing[24], instrumentation[29], etc.

Another rising trend in the field of virtualization is the use of VM based digital rights and copy protection. This trend has been growing in popularity over the last few years.

A generic and semi-automatic method for breaking VM based protection is proposed by Rolles [18]. He assumes that the VM is, broadly speaking, an infinite loop, with a large switch statement — the opcode dispatcher. Each case of this switch statement is a handler of a particular opcode.

At first, a reverse–engineer examines the virtual machine to construct a translator which maps the program instructions form the VM language (which might be stack–based) to some other language chosen by the engineer (which might be x86 assembly). Rolles calls the later language an intermediate representation (IR). The first step is done only once for a particular VM based protection. In the second step, the method extracts the VM opcode dispatcher and the obfuscated instructions from the executable. The opcodes of these instructions, however, don't have to correspond to those of the translator: the opcodes for every program instance can be permuted differently. So, in the third step, the method examines the dispatcher of the VM, and reveals the meaning of each opcode from the code executed by its handler. Finally, the obfuscated instructions are translated to the IR. At this point the program is not protected anymore, since it can be executed without the VM. Rolles further applies a series of optimizations to achieve a program which is close to the original one.

In this paper we don't try to obfuscate the VM: its source–code is publicly available and its detailed description appears herein. We protect the VM by holding secretly (inside the CPU) the opcode dispatcher. This makes it impossible to perform the second step described by Rolles.

Moreover, we claim that the security of the system can be guaranteed under the following assumptions:

- A chain of trust starting from the CPU can be realized.
- The inner state of the CPU can not be read by hardware.
- The CPU has a unique identifier stored in its register.

These assumptions are further refined throughout the paper.

The remainder of the paper is organized in the following way. Section II provides an overview of the related work. Section III outlines a step-by-step evolution of our system. Final details of the evolution are given in section IV. Section V describes the security of the proposed system and the performance impact is discussed in section VI. Appendix A provides an example of a program in C and the corresponding encrypted bytecode.

II. Related work

A. Virtual machines for copy protection

The two goal of introducing VM to trust computing are to encrypt and to obfuscate the program. Forcing the hackers to migrate from a familiar x86 environment to unfamiliar and obfuscated virtual environment, is intended to pose a greater challenge in breaking the software copy protection.

However, not much is published on the construction of virtual machines for digital rights protection as it would be counter productive to the obfuscation efforts that were the main reason for using VMs. Hackers, on the other hand, have opposite goals and tend to publish their results more often. Therefore, we often learn about VM protection from their breakers instead of their makers. For example, [20] is the

978-1-4577-0460-4/11/$26.00 ©2011 IEEE

most complete documentation of Code Virtualizer internals available outside Oreans.

Examples for using VM for copy protection include Cell OS level 1 in Sony Play station 3[25] (System Virtual Machine), Code Virtualizer[17] or VMProtect[30] (both are Process Virtual Machines).

In all cases very little was published by the software provider. However, the means for attacking virtual machine protection has been published by PS3 hackers [6] or by Rolles in [19] (with respect to VMProtect).

B. Hackers use of virtualization

By running protected software in a virtual environment it became possible to disguise it as a different system. For example, running OS X on standard hardware using a VM disguised as Apple machine (among others in [9].) Protection methods against virtualization were researched by [11]. Inspecting and freezing CPU and memory of a running system to obtain, for example, a copy of a copyrighted media, is another threat faced by media protectors such as NDS PC Show[16]. It is also possible to construct a malware that will obfuscate itself using a VM. [23] describes how malware obfuscated by VM can be detected.

C. Execution verification

Protecting and breaking software is a long struggle between vendors and crackers that began even before VM protection gained popularity.

Users demand mechanism to ensure the software they acquire is authentic. At the same time, software vendors require users to authenticate and to ensure the validity of the software license for business reasons.

Execution verification and signatures is now a part of Apple Mach-O[1] object format and Microsoft Authenticode[14]. It also exists as an extension to Linux ELF[15].

The trusted components have been heavily researched in industry [28] and academia (in [26] among others) with mixed results. Both software rights protectors and hackers can report partial success.

From hackers' camp, [27] is a fundamental paper dissecting all Microsoft's security mistakes in the first generation XBox.

III. System Evolution

In this section we describe the evolution of the proposed system in several phases — fictional interim versions of the system. For each version we describe a system and discuss its advantages, disadvantages and fitness to today's world of hardware and software components.

We explore means to improve the version and consider implementation details worth mentioning as well as any related work.

The first version, describes the system broken by Rolles. The second version cannot be broken (by Rolles) but has much stronger assumptions. The rest of the evolution process consists of our engineering innovation. The final version is as secure as the second one but requires only the assumptions of the first version.

A. Dramatis Personae

The following are actors that participate in our system use cases.

Hardware Vendor manufactures and provides the hardware. The Hardware Vendor can identify components he manufactured. The Hardware Vendor is trust worthy. (Possible real world example — Sony as Hardware Vendor of Play station 3).

Software Distributor distributes copy protected software. Interested in providing conditional access to the software.

In our case, the Software Distributor is interested in running the software on one End User CPU per license. (Possible real world example — VMProtect).

"Game" — The software we wish to protect. It may indeed be a computer game, a copyrighted video or other piece of software.

End User purchases at least one legal copy of the "Game" from the Software Distributor. The End User may be interested in providing other users with illegal copies of the "Game". The End User is not trustworthy.

Virtual Machine(VM) — A software component, developed and distributed by Software Distributor.

VM Interpreter — A sub–component of the Virtual Machine that interprets the instructions given by the "Game".

VM Compiler — A software component used by the software distributor to convert "Game" code developed in high level programming language to instructions interpretable by the VM Interpreter.

B. System Version 1

The VM Interpreter represents virtual, unknown ISA (Instruction Set Architecture) or permutation of a known instruction set (such as MIPS in our case). The VM Interpreter runs a loop:

Algorithm 1 System 1 - VM Interpreter run loop

```
while VM is running do
    Fetch next instruction
    Choose instruction handler routine
    Execute instruction using handler routine
end while
```

The VM Compiler reads a program in a high level programming language and produces output in the chosen ISA.

C. Discussion — System Version 1

Cryptographically speaking this is an implementation of replacement cipher on the instruction set. This method was described by Rolles[18] and used by VMProtect. Of course, the VM may include several other obfuscating subcomponents, that may even provide greater challenge to a malicious user, but this is beyond our scope. The protection is provided by the complexity of the VM and the user's lack of ability to understand it (and, as stated previously, in additional obfuscations).

Rolles describes a semi–automatic way to translate a program from the unknown ISA to intermediate representation

and later to the local machine ISA. Understanding how the VM works is based on understanding the interpreter. This problem is unavoidable. Even if the interpreter were implementing a secure cipher (such as AES) it wouldn't be able to provide a tangible difference, as the key to the cipher would also be stored in the interpreter in an unprotected form.

Therefore, it is vital to use a hardware component that the End User cannot reach to provide an unbreakable security.

D. System Version 2

The hardware vendor co-operates with the software distributor. He provides a CPU that holds a secret key, known to the software distributor. Furthermore, the CPU implements an encryption and decryption algorithms.

The compiler needs to encrypt the program with the CPU's secret key. This version doesn't require a VM (since the decryption takes place inside the CPU) and its operation is similar to that of a standard computer.

E. Discussion — System Version 2

This version could implement any cipher, including AES, which is considered strong. This form of encryption was described by Best [4]. Some information about the program, such as memory access patterns, can still be obtained.

This method requires manufacturing processors with cryptographic functionality (and secret keys) for decrypting every fetched instruction. Such processors are not widely available today.

F. System Version 3

This system is based on the system version 2 but the decryption algorithm is implemented in software. We alleviate the hardware requirements of system 2. The CPU stores a secret key which is also known to the software distributor. The VM Compiler reads the "Game" in high level programming language and provides the "Game" in an encrypted form where every instruction is encrypted using the secret key. The VM knows how to decrypt value stored in one register with a key stored in another register.

The VM Interpreter runs the following loop:

Algorithm 2 System 3 - VM Interpreter run loop

```
while VM is running do
    Read next instruction
    Decrypt the instruction
    Choose instruction handling routine
    Execute the instruction using handling routine
end while
```

G. Discussion — System Version 3

This version, is as secure as system version 2, assuming the VM internal state is stored inside the CPU cache memory at all times.

If only the VM runs on the CPU then we can make sure that the state of the VM (e.g., its registers), never leaves the CPU: the VM just has to access all the memory blocks incorporating its state, once in a while (the exact frequency depends on cache properties).

This method dramatically slows down the software — decrypting one instruction using AES takes up to 112 CPU cycles.

H. System Version 4

System 3 took dramatic performance hit which we now try to improve.

By combining System 1 and System 3 we implement a substitution cipher as in version 1. The cipher is polyalphabetic and special instructions embedded in the code define the permutation that will be used for the following instructions.

Similarly to system version 3 we use the hardware for holding a secret key that is known also to the software distributor.

The VM Interpreter runs the following code

Algorithm 3 System 4 - VM Interpreter run loop

```
while VM is running do
    Fetch current instruction
    Decrypt current instruction
    if Current instruction is not special instruction then
        Choose instruction handler routine
        Execute instruction using handler routine
    else
        Decrypt the instruction arguments using the private key
        Build new instruction permutation translation table
    end if
end while
```

I. Discussion — System Version 4

Section VI defines a structure of the special instructions and means to efficiently encode and reconstruct the permutations.

Dependent instructions should have the same arguments, as justified by the following example (extracted from the Pi Calculator (see Appendix A)):

```
01: $bb0_1:
02: lw $2, 24($sp)
03: SWITCH (X)
04: lw  $3, 28( $sp)
05: subu $2, $2, $3
06: beq $2, $zero, $bb0_4
07: ...
08: $bb0_3:
09: ...
10: lw $3, 20($sp)
11: SWITCH (Y)
12: div $2, $3
13: mfhi $2
14: SWITCH (Z)
15: sw $2, 36($sp)
16: $bb0_4:
17: sw $zero, 32($sp)
18: lw $2, 28($sp)
```

This is a regular MIPS code augmented with 3 special instructions on lines 3, 11 and 14. The extract consists of 3

basic blocks, labeled: bb0_1, bb0_3 and bb0_4. Note that we can arrive at the first line (line 17) of bb0_4 either from the conditional branch on line 6 or by falling through from bb0_3. In the first case line 17 is encoded by X and in the second case it is encoded by Z. The interpreter should be able to decode the instruction regardless of the control flow, thus X should be equal to Z. In order to characterize precisely the dependence between SWITCH instructions we introduce the following definition.

Definition [Flow] : Given a directed graph $G = (V, E)$ and a vertex $v \in V$ a flow is a pair (A_v, B_v) defined iteratively:
$v \in A_v$.
If $x \in A_v$ then for every $(x, y) \in E$, $y \in B_v$.
If $y \in B_v$ then for every $(x, y) \in E$, $x \in A_v$.
(No other vertices appear in A or B).

Claim: Two SWITCH instructions should share the same argument if and only if they are the last instructions of the basic blocks u and v s.t. $A_u = A_v$.

J. System Version 5

We rely on the previous version but give up on the assumption that the CPU is keeping a secret key that is known to the software distributor. Instead we run a key exchange algorithm [21].

Algorithm 4 Key exchange in system 5

The software distributor publishes his public key
The VM chooses a random number (the secret key); which is stored inside one of the CPU registers.
The VM encrypts it using a sequence of actions using the software distributor public key.
The VM sends the encrypted secret key to the software distributor.
The software distributor decrypts the value and gets the secret key.

K. Discussion — System Version 5

The method is secure if and only if we can guarantee the secret key was randomized in real (non–virtual) environment where reading CPU registers can be considered impossible. Otherwise it would be possible to run the program in a virtual environment where the CPU registers (and therefore, the secret key) are accessible to the user. Another advantage of random keys is that different "Game"s have different keys. Thus breaking one "Game" doesn't compromise the security of others.

L. System Version 6

This version is built on top of the system described in the previous section. Initially we run the verification methods described by Kennell and Jamieson in [11]. Using the methods described there we can guarantee the genuinity of the remote computer. i.e. the hardware is real and the software is not malicious.

A simple way to perform such verification is described below. In order to ensure that the hardware is real we can require any CPU to keep an identifier, a member of a random sequence. This will act as shared secret in the identification algorithm. The algorithm is performed by the VM without knowing the identifier itself. Identification algorithms are described in greater detail in [22].

In order to ensure the genuinity of the software we can initiate the chain of trust in the CPU itself (as in XBox 360). The CPU will initiate its boot sequence from internal, unreplaceable ROM.

M. Discussion — System Version 6

System 6 alleviates the risk of running inside a VM that is found in system version 5.

IV. Final Details

A. Scenario

In this section we provide a scenario involving all the dramatis personae. We have the following participants: Victor — the hardware vendor, Dan — the software distributor, Patrick — a programmer developing PatGame and Uma — an end user.

Uma has purchased a computer system supplied by Victor with Dan's VM preinstalled as part of the operating system. Patrick, who wants to distribute his "Game", sends it to Dan. Dan updates his online store to include this new "Game", PatGame.

Uma, who wants to play PatGame, sends a request for PatGame to Dan via his online store. Dan authenticates Uma's computer system (possibly in cooperation with Victor), as described in version 6. After the authentication completes successfully, Uma's VM generates a random secret key R, encrypts it with Dan's public key D and sends it to Dan. Dan decrypts the message obtaining R. This process was described in version 5. As described in version 4, Dan compiles PatGame with the key R and sends it to Uma. Uma's VM executes PatGame decrypting the arguments of special instructions with R.

A problem arises when Uma's computer is rebooted, since the key R is stored in a volatile memory. Storing it outside the CPU will compromise its secrecy and thus the security of the whole system. We propose to store the key R on Dan's side.

Suppose Uma wants to play an instance of PatGame already residing on her computer. Uma's VM generates a random secret key Q, encrypts it with Dan's public key D and send it to Dan. Dan authenticates Uma's computer. After the authentication completes successfully, Dan decrypts the message obtaining Q. Dan encrypts the stored key R with the key Q (using AES, for example) and sends it back to Uma. Uma decrypts the received message obtaining R, which is the program's decryption key. Thus the encrypted program doesn't have to be sent after every reboot of Uma's computer.

B. Compilation

Since the innovation of this paper is mainly the 4th version of the system, we provide here a more detailed explanation

of the compilation process. See Appendix A for an example program passing through all the compilation phases.

The compiler reads a program written in a high level programming language. It compiles it as usual up to the phase of machine code emission. The compiler, then, inserts new special instructions, which we call SWITCH, at random with probability p (before any of the initial instructions). The argument of SWITCH instruction determines the permutation applied on the following code (up to the next SWITCH instruction). Afterwards, the compiler calculates dependencies between the inserted SWITCH instructions. The arguments of the SWITCH instructions are set randomly but with respect to the dependencies.

The compiler permutes the instructions following SWITCH according to its argument. In the final pass we encrypt the arguments of all SWITCHes by AES with the key R.

C. Permutation

In order to explain how instructions are permuted, we should describe first the structure of the ISA we use — MIPS ISA. Every instruction starts with a 6-bit op-code, includes up to three 5-bit registers and, possibly, a 16-bit immediate value. The argument of the SWITCH instruction defines some permutation σ over 2^6 numbers and another permutation τ over 2^5 numbers. σ is used to permute the op-code and τ is used to permute the registers. Immediate values can be encoded either by computing them as described by Rolles [18] or by encrypting them using AES.

V. Security

The method described by Rolles, requires a complete knowledge of the VM's interpreter dispatch mechanism. This knowledge is essential for implementing a mapping from bytecode to intermediate representation (IR). In the described system, a secret key, which is part of the dispatch mechanism, is hidden from an adversary. Without the secret key, the permutations are constructed secretly. Without the permutations the mapping from bytecode to IR can not be reproduced.

The described compiler realizes an auto–key substitution cipher. This class of ciphers is on one hand more secure than the substitution cipher used by VMProtect, and, on the other hand, doesn't suffer from the performance penalties typical to the more secure AES algorithm.

As discussed by Goldreich [8] some information can be gathered from memory accesses during program execution. The author proposes a way to hide the access patterns, thus not letting an adversary to learn anything from the execution.

VI. Performance

In this section we analyze in detail the performance of the proposed cipher. Throughout the section we compare our cipher to AES. This is due to recent advances in CPUs that make AES to be the most appropriate cipher for program encryption [10]. We provide both theoretical and empirical observations proving our algorithm's supremacy.

We denote the number of cycles needed to decrypt one word of length w using AES by α.

A. Version 3 Performance

Consider version 3 of the proposed system and assume it uses AES as a cipher. Every instruction occupies exactly one word, so n instructions can be decrypted in $n\alpha$ cycles.

B. Switch Instructions

As described above, the switch instruction is responsible for choosing the current permutation σ. This permutation is, then, used to decrypt the op-codes of the following instructions.

Some details were omitted previously, since they affect only system's performance, but not its security or overall design:

- How does the argument of a SWITCH instruction encode the permutation?
- Where is the permutation stored inside the processor?

Before answering these questions we introduce two definitions:

- Given an encrypted program, we denote the set of all instructions encrypted with σ by I_σ and call it color–block σ.
- Given a set I of instructions, we denote by $D(I)$ the set of different instructions (those having different op–codes) in I.

The key observation is that it is enough to define how σ acts on the op–codes of $D(I_\sigma)$ — instructions occurring in the color–block σ. Likewise, we noticed that some instructions are common to many color–blocks, while others are rare.

Denote by $F = \{f_1, f_2, \ldots, f_\ell\}$ the set of the ℓ most frequent instructions. Let $R(I)$ be the set of rare instructions in I, i.e. $R(I) = D(I) - F$. The argument of SWITCH preceding a color–block σ has the following structure (and is encrypted by AES as described in version 5):

$$\sigma(f_1), \sigma(f_2), \ldots, \sigma(f_\ell),$$
$$r_1, \sigma(r_1), r_2, \sigma(r_2), \ldots, r_k, \sigma(r_k)$$

where $R(I_\sigma) = \{r_1, r_2, \ldots, r_k\}$. If σ acts on m–bits long op–codes, then the length of σ's encoding is $\phi = (\ell + 2k)m$ bits. Thus, it's decryption takes $\frac{(\ell+2k)m}{w}\alpha$ cycles.

C. Version 4 Performance

Consider a sequence of instructions between two SWITCHes in the program's control flow. Suppose these instructions belong to I_σ and the sequence is of length n. The VM Interpreter starts the execution of this sequence by constructing the permutation σ. Next, the VM Interpreter goes over all the n instructions, decrypts them according to σ and executes, as described in version 5.

The VM Interpreter stores $\sigma(f_1), \sigma(f_2), \ldots, \sigma(f_\ell)$ in the CPU registers and the rest of σ — in the cache. This allows, decryption of frequent instructions in one cycle. Decryption of rare instructions takes $\beta + 1$ cycles where β is the cache latency (in cycles). Denote by q the number of rare instructions in the sequence.

ℓ	0	1	2	3	4	5	6
ϕ	58	39	42	41	58	46	48
$\frac{q}{n}$	100%	70%	50%	34%	34%	26%	21%

Fig. 1. Correlation between ℓ, ϕ and q. Here $p = 0.2$

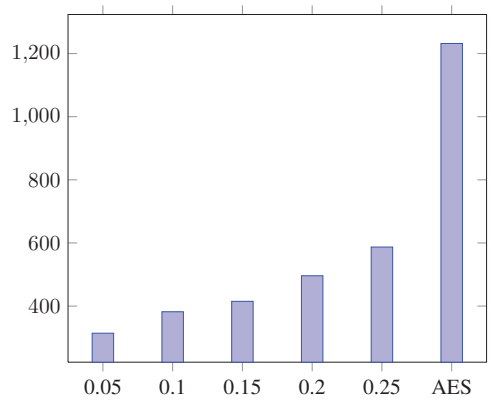

Fig. 2. Program decryption time (in cycles) using AES and our cipher with different values of p. $\ell = 4$. $\alpha = 14$.

We are, now, ready to compute the number of cycles needed to decrypt the sequence:

$$\begin{aligned} \sigma \text{ construction:} \quad & \frac{(\ell + 2k)m}{w}\alpha + \\ \text{frequent instructions:} \quad & (n - q) + \\ \text{rare instructions:} \quad & q(\beta + 1) \end{aligned}$$

D. Comparison

On MIPS op-codes are $m = 6$ bits long and $w = 32$. The best available results for Intel newest CPUs argue that $\alpha = 14$ [32]. Typical CPUs' Level–1 cache latency is $\beta = 3$ cycles. The correlation between ℓ, ϕ and q is depicted on figure 1.

We have noticed that most of the cycles are spent constructing permutations, this is done every time SWITCH is encountered. The amount of SWITCHes, and thus the time spent constructing permutations, varies with the probability p of SWITCH insertion. The security, however, varies as well. Figure 2 compares program decryption time (in cycles) using AES and our solution with different values of p.

Note that on CPUs not equipped with AES/GCM [32], like Pentium 4, $\alpha \geqslant 112$. In this case our solution is even more beneficial. Figure 3 makes the comparison.

VII. Acknowledgments

We would like to thank Dmitry Sotnikov for his brilliant ideas and his valuable comments on both form and content of this paper. This work would not have been possible without his help.

Nezer J. Zaidenberg work was financed by COMAS graduate school add Artturi and Ellen Nyyssönen foundation.

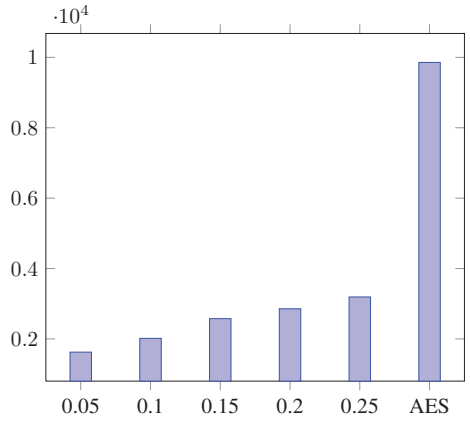

Fig. 3. Program decryption time (in cycles) using AES and our cipher with different values of p. $\ell = 4$. $\alpha = 112$.

VIII. Summary

We discussed several steps toward software protection. Our paper didn't discuss obfuscation procedures (beyond using a VM) as very little can be said about. Since obfuscations may provide the bread and butter of many real life products and since such measures may make the challenges faced by software hackers much more difficult we make no claim regarding any product vulnerability.

References

[1] Apple Ltd. Mac OS X ABI Mach-O file format reference. http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html.

[2] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *In SOSP (2003)*, pages 164–177, 2003.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *FREENIX Track: 2005 USENIX Annual Technical Conference*, 2005.

[4] R. M. Best. Preventing software piracy with crypto-microprocessors. In *Proceedings of IEEE Spring COMPCON 80*, pages 466–469, February 1980.

[5] D. Box. *"Essential .NET, Volume 1: The Common Langugage Runtime"*. Addison-Wesley, 2002.

[6] bushing, marcan, and sven. Console hacking 2010 ps3 epic fail. In *CCC 2010: We come in peace*, 2010.

[7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *5th USENIX Symposium on Network Systems Design and Implementation*, 2008.

[8] O. Goldreich. Toward a theory of software protection. In *Proceedings of advances in cryptology – CRYPTO86*, 1986.

[9] A. Graf. Mac OS X in KVM. In *KVM Forum 2008*, 2008.

[10] Intel. Breakthrough AES performance with Intel AES NewInstructions, 2010.

[11] R. K. . L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[12] A. Kivity. Kvm: The kernel-based virtual machine. In *Ottawa Linux Symposium*, 2007.

[13] T. Lindoholm and F. Yellin. *"The Java Virtual Machine Specification, 2nd ed."*. Addison–Wesley, 1999.

[14] Microsoft Cooperation. Windows authenticode portable executable signature form. http://msdn.microsoft.com/en-us/windows/hardware/gg463180.aspx.

[15] millerm. elfsign. http://freshmeat.net/projects/elfsign/.

[16] NDS. PC Show. http://www.nds.com/solutions/pc_show.php.
[17] Oreans Technologies. Code virtualizer. http://www.oreans.com/products.php.
[18] R. Rolles. Unpacking virtualization obfuscators. In *Proc. of 4th USENIX Workshop on Offensive Technologies (WOOT '09)*, 2009.
[19] R. E. Rolles. Unpacking VMProtect. http:/http://www.openrce.org/blog/view/1238/.
[20] scherzo. Inside code virtualizer. http://rapidshare.com/files/16968098/Inside_Code_Virtualizer.rar.
[21] B. Schneier. Key–exchange algorithms. In *Applied Cryptography 2nd ed.*, chapter 22. Wiley, 1996.
[22] B. Schneier. Key–exchange algorithms. In *Applied Cryptography 2nd ed.*, chapter 21. Wiley, 1996.
[23] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proc. of the 30th IEEE Symposium on Security and Privacy*, 2009.
[24] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of 16th ACM Conference on Computer and Communications Security*, November 2009.
[25] SONY Consumer Electronics. Playstayion 3. http://us.playstation.com/ps3/.
[26] M. Srivatsa, S. Balfe, K. G. Paterson, and P. Rohatgi. Trust management for secure information flows. In *Proceedings of 15th ACM Conference on Computer and Communications Security*, October 2008.
[27] M. Steil. 17 mistakes Microsoft made in the Xbox security system. In *22nd Chaos Communication Congress*, 2005.
[28] Trusted Computing Group. TPM main specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.
[29] Valgrind's developers. Valgrind. http://www.valgrind.org.
[30] VMPSoft. Vmprotect. http://www.vmprotect.ru.
[31] VMWare. Vmware. http://www.vmware.com.
[32] Wikipedia, the free encyclopedia. AES instruction set.

APPENDIX A

This appendix provides a detailed example of a program passing through all the compilation phases. The original program is written in C. It computes the sum of the first 800 digits of π.

```
int main(){
  int a=10000,b,c=2800,d,e,f[2801],g,s;

  for(;b-c;) f[b++]=a/5;

  for(;d=0,g=c*2;c-=14,e=d%a){
    for(b=c;
      d+=f[b]*a,f[b]=d%--g,d/=g--,--b;
          d*=b);
        s += e+d/a;
  }

  return s;
}
```

The corresponding output of the compiler is listed below. It is a combination of MIPS assembly and MIPS machine instructions. The leftmost column contains the instruction number. The second column contains the machine instruction. The rightmost column contains the assembly instruction. Block labels are emphasized.

Instructions of the form

```
addiu $zero, $zero, ...
```

are an implementation of the SWITCH instruction. Colors correspond to the permutation that should be applied to the instructions.

For example, consider the instruction at 000c. It sets the current permutation to 5 (corresponding to gray color). As a result, all the following instructions (up to the next SWITCH) are colored gray. Note that the instruction at 000c is not colored gray, since it should be encoded by the previous permutation (pink, corresponding to number 4 and set at 0005).

After the instructions are permuted according to their colors, the final phase takes place: the compiler encrypts special instructions' arguments using AES (with the secret key).

$main (1)

```
0001 24000002 addiu $zero, $zero, 2

0002 24000003 addiu $zero, $zero, 3

0003 27bdd400 addiu $sp, $sp, -11264
0004 24022710 addiu $2, $zero, 10000
0005 24000004 addiu $zero, $zero, 4

0006 afa00010 sw $zero, 16($sp)
0007 24030af0 addiu $3, $zero, 2800
0008 afa20014 sw $2, 20($sp)
0009 afa3001c sw $3, 28($sp)
```

$bb0_1 (4)

```
000b 8fa20018 lw $2, 24($sp)
000c 24000005 addiu $zero, $zero, 5

000d 8fa3001c lw $3, 28($sp)
000e 00431023 subu $2, $2, $3
000f 10020000 beq $2, $zero, $bb0_4

0011 3c026666 lui $2, 26214
0012 34426667 ori $2, $2, 26215
0013 8fa30014 lw $3, 20($sp)
0014 8fa40018 lw $4, 24($sp)
0015 00620018 mult $3, $2
0016 00001010 mfhi $2
0017 00400803 sra $3, $2, 1
0018 24000006 addiu $zero, $zero, 6

0019 0040f801 srl $2, $2, 31
001a 27a50030 addiu $5, $sp, 48
001b 00801000 sll $6, $4, 2
001c 24840001 addiu $4, $4, 1
001d 24000004 addiu $zero, $zero, 4

001e 00621021 addu $2, $3, $2
001f 00a61821 addu $3, $5, $6
0020 afa40018 sw $4, 24($sp)
0021 ac620000 sw $2, 0($3)
0022 0800000a j $bb0_1
```

$bb0_3 (7)

```
0024 8fa20020 lw $2, 32($sp)
0025 24000008 addiu $zero, $zero, 8

0026 8fa30014 lw $3, 20($sp)
0027 0043001a div $2, $3
0028 00001012 mflo $2
```

```
0029 8fa30024 lw $3, 36($sp)
002a 8fa42bf8 lw $4, 11256($sp)
002b 00621021 addu $2, $3, $2
002c 00821021 addu $2, $4, $2
002d afa22bf8 sw $2, 11256($sp)
002e 8fa2001c lw $2, 28($sp)
002f 2442fff2 addiu $2, $2, -14
0030 afa2001c sw $2, 28($sp)
0031 8fa20020 lw $2, 32($sp)
0032 8fa30014 lw $3, 20($sp)
0033 24000009 addiu $zero, $zero, 9

0034 0043001a div $2, $3
0035 00001010 mfhi $2
0036 24000005 addiu $zero, $zero, 5

0037 afa20024 sw $2, 36($sp)
```

$bb0_4 (5)

```
0039 afa00020 sw $zero, 32($sp)
003a 8fa2001c lw $2, 28($sp)
003b 00400800 sll $2, $2, 1
003c afa22bf4 sw $2, 11252($sp)
003d 10020000 beq $2, $zero, $bb0_8

003f 8fa2001c lw $2, 28($sp)
0040 afa20018 sw $2, 24($sp)
```

$bb0_6 (5)

```
0042 8fa20018 lw $2, 24($sp)
0043 27a30030 addiu $3, $sp, 48
0044 00401000 sll $2, $2, 2
0045 00621021 addu $2, $3, $2
0046 2400000a addiu $zero, $zero, 10

0047 8c420000 lw $2, 0($2)
0048 8fa40014 lw $4, 20($sp)
0049 8fa50020 lw $5, 32($sp)
004a 00440018 mult $2, $4
004b 2400000b addiu $zero, $zero, 11

004c 00001012 mflo $2
004d 00a21021 addu $2, $5, $2
004e afa20020 sw $2, 32($sp)
004f 8fa42bf4 lw $4, 11252($sp)
0050 2484ffff addiu $4, $4, -1
0051 afa42bf4 sw $4, 11252($sp)
0052 8fa50018 lw $5, 24($sp)
0053 00a01000 sll $5, $5, 2
0054 0044001a div $2, $4
0055 00001010 mfhi $2
0056 00651821 addu $3, $3, $5
0057 ac620000 sw $2, 0($3)
0058 8fa22bf4 lw $2, 11252($sp)
0059 2400000c addiu $zero, $zero, 12

005a 2443ffff addiu $3, $2, -1
005b afa32bf4 sw $3, 11252($sp)
005c 8fa30020 lw $3, 32($sp)
005d 0062001a div $3, $2
005e 00001012 mflo $2
005f afa20020 sw $2, 32($sp)
0060 24000007 addiu $zero, $zero, 7

0061 8fa20018 lw $2, 24($sp)
0062 2442ffff addiu $2, $2, -1
0063 afa20018 sw $2, 24($sp)
0064 10020000 beq $2, $zero, $bb0_3

0066 8fa20018 lw $2, 24($sp)
0067 2400000d addiu $zero, $zero, 13

0068 8fa30020 lw $3, 32($sp)
0069 00620018 mult $3, $2
006a 00001012 mflo $2
006b 24000005 addiu $zero, $zero, 5

006c afa20020 sw $2, 32($sp)?006d 08000041 j
$bb0_6
```

$bb0_8 (5)

```
006f 8fa22bf8 lw $2, 11256($sp)
0070 27bd2c00 addiu $sp, $sp, 11264
0071 03e00008 jr $ra
```

APPENDIX B

The code of Truly–protect is freely available at
http://code.google.com/p/truly-protect

PVI

SYSTEM FOR EXECUTING ENCRYPTED JAVA PROGRAMS

by

Kiperberg, M.; Resh, A.; Algawi, A.; Zaidenberg, N.J. Submitted

IEEE Transactions on Dependable and Secure Computing

System for Executing Encrypted Java Programs

Michael Kiperberg, Amit Resh, Asaf Algawi, and Nezer Zaidenberg

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, it is proven 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. Adequate systems for managing secret keys in a protected trust-zone and supporting execution of encrypted native code have been proposed in the past [1], [2]. Nevertheless, these systems are not suitable as is for protecting managed code. In this paper we propose enhancements to these systems so they support execution of encrypted Java programs that are resistant to reverse engineering. The main difficulty underlying Java protection with encryption is the interpretation that is performed by the JVM. The JVM will require the key to decrypt the encrypted portions of Java code and there is no feasible way of securing the key inside the JVM. To solve this, the authors propose implementing a Java bytecode interpreter inside a trust-zone, governed by a thin hypervisor. This interpreter will run in parallel to the standard JVM, both cooperating to execute encrypted Java programs.

Index Terms—Java, Trusted Computing, Hypervisor, Virtualization, Remote Attestation

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" [3], [4]. The term obfuscation refers to making software instructions difficult for humans to understand by deliberately cluttering the code with useless, confusing pieces of additional software syntax or instructions. However, even when changing the software code and making it obfuscated, the content is still readable to the skilled hacker [5], [6].

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 [7], [8], [9], 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 [10], [11].

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 recent years, programs that are targeted at managed execution environments have become widespread [12]. Unlike regular (native) programs, managed programs cannot be executed directly by the CPU and therefore require a special (native) program to interpret the managed program. Managed execution environments are superior to native environments in memory management, debugging, and profiling support. All of these benefits increased the popularity of managed execution environments among developers of desktop and mobile applications.

We would like to stress the key difference between native and managed execution environments. While it is possible to guarantee that a sequence of native instructions cannot be intercepted (read or modified) during its execution by a CPU, such a guarantee cannot be made for a managed execution environment, since an unexpected behavior can be introduced into the software that implements the managed execution environment. There is, therefore, a need for a technique for executing safely encrypted managed programs on the available managed execution environments.

In this paper, we present a system that allows encrypting and executing programs written for the Java Virtual Machine (JVM) [13]. The system execution engine is based on a thin hypervisor and works in cooperation with the JVM through the standardized JVM Tool Interface (JVM TI) and the Java Native Interface (JNI). The hypervisor acquires the decryption key during the initialization of the JVM and is responsible for decrypting and executing the encrypted parts of the Java program. The unencrypted parts of the program are executed directly by the JVM, which, in many cases, makes the overall performance comparable to the

• *M. Kiperberg, A. Resh, A. Algawi and N. Zidenberg are with the Department of Mathematical Information Technology, University of Jyvaskyla, Jyvaskyla, Finland. E-mails: michael@trulyprotect.com, amit@trulyprotect.com, asaf@trulyprotect.com, nezer@trulyprotect.com*

performance of the original, unencrypted, program.

The paper is organized as follows. Sections 2 and 3 describe Java's instruction set and its executable file format. Section 4 presents an API which allows the inspections and manipulation of Java programs and their execution environments. Hypervisors are discussed in section 5. We present the design of our system in section 6 and discuss a specific aspect of this design in section 7. The performance and security measurements of the system are presented in section 8. Section 9 discusses the limitations of the presented system and its possible extensions. Section 10 summarizes the results of this paper.

2 Java Bytecode

Java bytecode is the instruction set of the JVM [13]. Programs written in Java are compiled to the Java bytecode and stored, together with additional information, in class files.

JVM is a stack machine: the arguments of an instruction are pushed onto a stack, the instruction is executed, which pops the arguments off the stack and the result is pushed back onto the stack. Moreover, JVM is a strongly typed machine, which means that each value on the stack is tagged with a type. Each instruction verifies that the types of its arguments are valid, and failure during this verification results in an exception condition.

Each instruction consists of a one-byte opcode and a list of arguments, whose length is determined by the opcode (in most cases). Currently 198 opcodes are in use, which can be subdivided into a number of broad groups:

1) Load and store (push the value of a local variable onto a stack)
2) Arithmetic and logic (pop two values off the stack and push their sum onto the stack)
3) Type conversion (change the type of the value at the top of stack from integer to double)
4) Object creation and manipulation (create an instance of type T)
5) Operand stack management (duplicate the value at the top of the stack)
6) Control transfer (pop the value off the stack and if it is true increment the IP by 42)
7) Method invocation and return (terminate the current method)
8) Other (throw an exception)

Many instructions have prefixes and suffixes that determine the types of the operands they operate on.

Each method has its own stack and its own area of local variables (which also includes method's parameters). Any constants used in a method, e.g. numerical constants, type names, or method names, are stored in a constant pool belonging to the class in which the method is defined. The method references these constants via their indices.

Many languages, Java among them, have introduced a notion of exceptions. An exception is an abnormal condition detected by the program, which cannot be handled locally, i.e. in the method which detected this condition. The process of transferring the control to the handler of the abnormal condition is called *throwing an exception*. The handler is called an *exception handler*. The exception handler usually needs additional information about the nature of the abnormal condition. In Java this information is encapsulated in an object whose type implements the interface *Throwable*.

Most exceptions occur synchronously as a result of an action by the thread in which they occur. An asynchronous exception, by contrast, can potentially occur at any point in the execution of a program. Asynchronous exceptions are not covered in our work. Synchronous exceptions can occur as a result of execution of the *athrow* instruction or any other instruction that specifies an exception as a possible result. For example, the *idiv* instruction, which divides two integers, throws an ArithmeticException if the value of the divisor is 0.

Each method may be associated with zero or more exception handlers. An exception handler specifies the range of instructions for which the exception handler is active, and describes the type of exception that the exception handler is able to handle. When an exception is thrown, the JVM searches for a matching exception handler in the current method. If a matching exception handler is found, the system branches to the exception handling code specified by the matched handler.

If no such exception handler is found in the current method, the current method's frame is popped, reinstating the frame of the invoking method. The exception is then rethrown in the context of the invoker's frame and so on, continuing up the method invocation chain.

3 Java File Structure

Java is an object-oriented programming language [14]. All code in a Java program is written in classes. The source code is compiled into intermediate bytecode that is stored in *class files* [13]. Each *class file* contains the compiled bytecode of a class along with descriptions of its fields, interfaces and methods.

A *class file* is a serialized stream of 8-bit bytes, grouped as 1, 2, 4 and 8 byte items. The file contains information fields and several major sections:

- constant pool
- fields
- methods
- attributes

The *constant pool* is a table in which the entries contain: numerical constants, string literals, class and interface names, field names as well as some additional constants referred to inside the file.

The *methods* section is a table in which the entries contain all the information relevant to the methods of the class. Most importantly, it contains the Java bytecode of the compiled method.

Java applications normally constitute an abundance of *class files*, metadata and a variety of resource files such as icons, text files and images. The files are usually organized in a directory structure. An entire Java application subdirectory structure along with all its associated files can be packaged in a single *jar file*. The *jar file* is an archive file, based on ZIP format compressed files. It provides a convenient method to distribute complete Java applications. The *jar file* is also used to store a Java library.

The proposed system for encrypting Java programs is designed to encrypt all *class files* within a given *jar archive*. A configuration file can be specified to include or exclude given sets of *class files* based on file names or sub-directory location.

Similarly to other instrumentation tools [15], [16], [17], our encryption tool analyzes each *class file* that was defined to be protected. It analyzes and makes appropriate changes to the file by overwriting *nop*s on the original bytecodes in the class' methods and storing the encrypted bytecodes of each method in a separate section in the *class file*.

Once encryption is completed, the encryption tool serializes the class structures back into a modified *class file*, which replaces the original one in the *jar file*.

4 JNI/JVM TI

JVM TI [18] is an application programming interface (API) provided by the JVM that allows the inspection and the controlling of the state of the JVM and the program it executes. This API is usually used by performance profilers and debuggers [19], [20], [21], [22]. JVM TI is a two-way interface. A client of JVM TI, an *agent*, can be notified of interesting occurrences through events. An agent can query the JVM through many functions, either in response to events or independent of them. An agent is realized in a dynamic library, a dynamic link library on Windows or a shared object on Linux. In order to attach an agent to a JVM, its path should be passed on the command line during the invocation of the JVM.

The JVM loads all agents before loading any classes. For each loaded agent, the Agent_OnLoad function is called. This function is responsible for specifying the set of events to be intercepted and the interception functions. The interception functions may have different sets of parameters that capture the parameters of event occurrences. Consider the two events ThreadStart and ClassPrepare. The ThreadStart event occurs whenever the JVM creates a new thread and the interception function receives, as a parameter, the identifier of the newly created thread. The ClassPrepare event occurs whenever a class is loaded and the interception function receives, as a parameter, the identifier of the thread which loaded the identifier of the loaded class.

In addition to events interception, JVM TI allows an agent to inspect and manipulate the state of the JVM and the state of the program it executes. For example, the GetClassMethods function retrieves the method identifiers of a specified class. The methods can be further investigated using functions like GetMethodName, which retrieves the name and the signature of a specified method, and GetBytecodes, which retrieves the bytecodes of a specified method.

Another family of JVM TI functions allows inspection of dynamic aspects of the execution. This family includes functions such as GetLocalVariable, which retrieves the value of method's local variable (in Java parameters are also variables), and GetStackTrace, which retrieves information about the callers of the current method.

Finally, another family of JVM TI functions allows modifying the state of the program. This family includes functions such as SetLocalVariable, which assigns a value to method's local variable, SetBreakPoint, which sets a breakpoint at a specified location of a specified method, and ForceEarlyReturnObject, which requests to terminate the execution of the current method.

JNI is an API provided by the JVM that enables Java programs to call and be called by native programs. JNI provides functions that can inspect and manipulate Java objects. These functions can be subdivided to the following families: class operations, exceptions, accessing fields, calling methods, etc.

The class operations family includes functions such as FindClass, which loads a class by its name, and IsAssignableFrom, which determines whether an object of one class can be safely cast to another class. The exceptions family includes functions such as Throw and ThrowNew, that request to handle the specified exception, and ExceptionOccurred, which determines whether an exception is being handled. The accessing fields family includes functions such as GetObjectField, which retrieves the value of the specified field in the specified object, and SetObjectField, which assigns a value to the specified field in the specified object. The calling methods family includes functions such as CallVoidMethod, which calls the specified method of the specified object, and CallNonvirtualVoidMethod, which calls the specified method of the specified class (not necessarily object's class).

The JVM passes pointers to the JVM TI and the JNI on each invocation of the interception functions. The interception functions can use these interfaces to manipulate the state of the program and the JVM. In particular, these interfaces provide means to build a partial interpreter, i.e. an interpreter of some but not all of the code, inside a JVM TI agent. Such an interpreter is required to cooperate with the JVM, so that the JVM can observe the modification in state made by the interpreter and vice-versa. The functions provided by JVM TI and JNI are sufficient to implement this cooperation, as will be explained later.

Two important events provided by JVM TI with regard to exceptions are: Exception and ExceptionCatch. The first event occurs when an exception is thrown due to *athrow* execution or any other instruction. The interception function receives the location in which the exception was thrown, the exception object, and the location of the exception handler which was found by the JVM. The ExceptionCatch event occurs after the control was transferred to the exception handler but before its first instruction was executed. The interception function receives the location of the exception handler and the exception object. The system described in this paper uses only the ExceptionCatch event. The Exception event was described for completeness.

5 Thin Hypervisor

A hypervisor, also referred to as a Virtual Machine Monitor (VMM), is software, which may be hardware-assisted, to manage multiple virtual machines on a single system [23]. 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

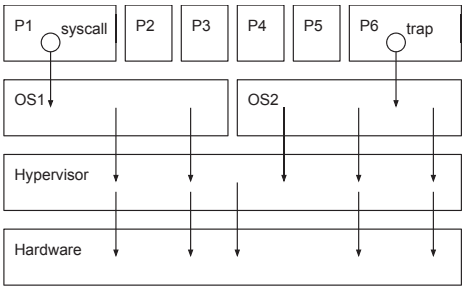

Fig. 1. Virtualized system featuring a hypervisor and two operating systems executing six programs. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions and other interrupts, transfer control from user mode applications to their operating system.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

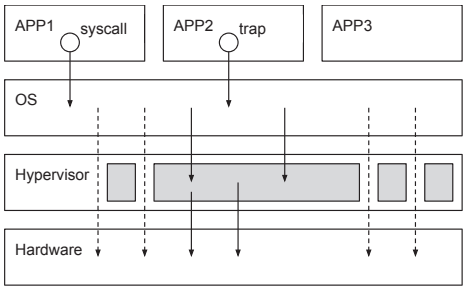

Fig. 2. Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service from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

unaccompanied on the entire hardware platform. The hypervisor is referred to as the *host*, while the virtual machines are referred to as *guests*. Hypervisors are further categorized as: type-1 (or bare metal) and type-2 hypervisors. A type-1 hypervisor executes independently and directly over the system hardware. The OS of the guests run above the hypervisor, in effect decoupled by the hypervisor from the system hardware. A type-2 hypervisor [24] executes above a cooperating OS, where guests run atop the hypervisor. This type of hypervisor uses the cooperating OS as a means to access and manage hardware resources.

Hypervisors have been in use from as early as the 1960s on IBM mainframe computers [25]. After 2005, Intel and AMD introduced hardware support for virtualization (Intel VT-X [26], AMD AMD-V [27]) which allowed implementing type-1 hypervisors in the ubiquitous PC platforms.

In order to support multiple OS guests, a type-1 hypervisor must unobtrusively intercept OS access to hardware resources so it can attend to them itself [28]. The hypervisor can then manage hardware allocations that maintain proper separation between the guests. The guest OS is unaware of the hypervisor's intervention, as it experiences a normal hardware access cycle. The only distinction is the elapsed time, since the hypervisor mediation has a time-toll. This property led to a debate regarding the detectability of a hypervisor [29], [30], [31], [32].

To intercept OS hardware access, hypervisors are configured to intercept privileged instructions, memory access, interrupts, exceptions and I/O, which are the OS vehicles for hardware access. Executing an intercepted privileged instruction causes a hypervisor VM_EXIT. In other words, the guest VM is exited and the configured Hypervisor intercept-routine is executed. When this occurs, the CPU mode changes from guest-mode to host-mode. Guest applications that require hardware resources execute system calls to request support from their OS. Figure 1 depicts this chain-of-execution for a type-1 hypervisor with two guest stacks. After fulfilling the intercept, the hypervisor indiscernibly returns to the guest. While hypervisors were generally designed to serve as virtual machine monitors, type-1 hypervisors, which control the underlying hardware platform, are also very good candidates to serve as software security facilitators.

The authors propose the use of a type-1 hypervisor environment for securing a single guest stack. Rather than wholly virtualizing the hardware platform, a special breed of hypervisor, called a thin-hypervisor, is used [33], [34]. The thin-hypervisor is configured to intercept only a small portion of the system's privileged events. All other privileged instructions are executed without interception, directly, by the OS. The thin-hypervisor only intercepts the set of privileged instructions that allows it to protect an internal secret (such as cryptographic key material) and protect itself from subversion. Figure 2 depicts a thin-hypervisor supporting a single guest stack. Since the thin-hypervisor does not control most of the OS interaction with the hardware, multiple OS are not supported. However, system performance is kept at an optimum.

A Hypervisor facilitates a secure environment by:

- setting aside portions of memory that can be accessed only when the CPU is in host mode
- storing cryptographic key material in privileged registers and
- intercepting privileged instructions that may compromise its protected memory or key material

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

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

In the proposed system, to execute encrypted Java bytecode, the thin-hypervisor capabilities are exploited to decrypt the encrypted Java bytecode (using the secret key) into

protected memory regions and following up with interpretation and execution of the decrypted instructions while in host mode.

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

The authors' approach to solving these questions is comprised of the following principles:

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

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

The vehicle to perform this remote verification is a piece of code, called an attestation-challenge [1], [35], [36], [37]. The attestation-challenge is administered by the key-server to the remote machine, as it is configuring the thin-hypervisor. The thin-hypervisor is required to load and execute the challenge code, returning an attestation result to the key-server within a limited time-frame. The attestation-challenge calculates the checksum of the thin-hypervisor code, but in addition mangles the checksum calculation with hardware-driven side-effects, sampled by the challenge as it is executing. The side-effect samples are hardware-generated counts of hardware events, such as cache hits or misses, TLB hits or misses etc.

The key-server considers a correct response received within the allotted time-frame proof that the correct thin-hypervisor code is executing and has true control of the remote system's hardware.

6 System Design

The system we present comprises four main components: (1) encryption tool, (2) JVM TI agent, (3) thin hypervisor, (4) attestation server. Figure 5 depicts the relationship between these components.

The encryption tool processes each *class file* by first deserializing it into memory based structures. The code bytes of each method are located and zeroed out to create a sequence of *nop* instructions except for the very first code byte and the last three bytes. In the first code-byte (offset 0) it always inserts an *aconst_null* opcode (a single-byte instruction that pushes a NULL on the operand stack). In the last 3 bytes it inserts a *jump* instruction that loops back to the 1st *nop* instruction (offset 1). The reason for this pattern is to allow a means to synchronize the JVM processing of the decrypted bytecode, as will be detailed later. The *aconst_null* opcode at the beginning of the method's code is required to appease the Java verifier. Without the *aconst_null*, the verifier contemplates that in the event of exception handling, the loop back to the method's start may have a stack depth of 1, while during other loops, stack depth is 0. This discrepancy is not allowed. With the *aconst_null* opcode, the stack depth is always 1 regardless, and thus allowed by the verifier. Figure 3 illustrates this process.

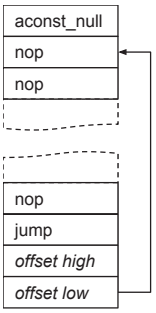

Fig. 3. Pattern of an encrypted method.

Methods that are smaller than 5 bytes and cannot accommodate this pattern are simply not encrypted. The authors assume that the security penalty for this will be insignificant. If there exist (rare) very short methods that must be encrypted, they will need to be artificially enlarged with do-nothing instructions.

The encryption tool extends the existing constant pool to make room for encrypted versions of protected methods' bytecode. The original bytecodes of each method are encrypted and inserted in a new record appended at the end of the constant pool table.

After adding all the new encrypted entries, the encryption tool adds a trailer record at the very end, detailing the number of preceding encrypted entries. When the Java class is loaded for execution, the runtime decryption and execution engine can find this information by looking up the trailer-record at the end of the constant pool.

Once encryption is completed, the encryption tool serializes the class structures back into a modified *class file*, which replaces the original one in the *jar file*.

During the initialization of the JVM TI agent, it deploys the hypervisor and installs interception functions for the following events (1) class loading, (2) breakpoint, (3) exception catch. The class loading event occurs whenever the JVM loads a class and before any of the class code is executed. Upon this event the agent inspects the class and determines whether it is encrypted. If so, the agent installs a breakpoint at the first instruction of each method. These breakpoints induce a breakpoint event on each entry to the encrypted methods. The agent intercepts the breakpoint event, resolves the method that hosts the hit breakpoint, and begins the interpretation process.

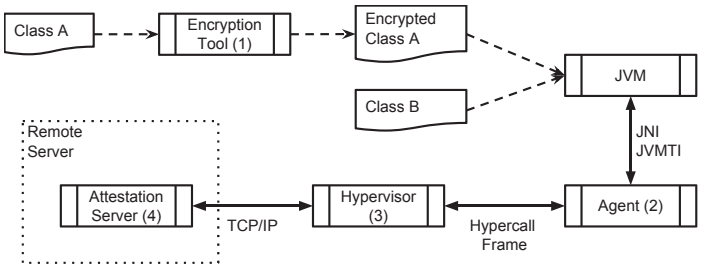

Fig. 5. Relationship between the different components of the described system. The encryption tool (1) transforms regular Java classes into encrypted ones. Regular and encrypted Java classes are then loaded by the JVM. The JVM loads a JVM TI agent (2) through a JVM TI interface. The agent links the hypervisor to the JVM and assists in the interpretation process. The agent communicates with the JVM through JVM TI and JNI. The communication between the agent and the hypervisor is based on hypercalls and execution frames. The hypervisor receives the decryption key from a remote server, which attests the validity of the hypervisor and the hardware on which it executes.

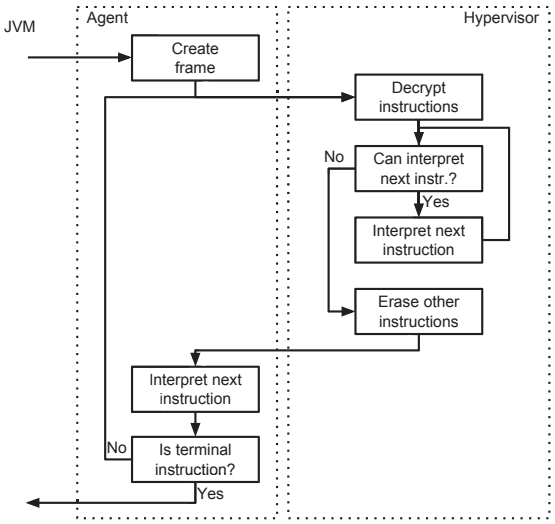

Fig. 6. A simplified control flow during encrypted method execution. The JVM reaches the breakpoint installed by the agent, and transfers the control to the JVM TI agent. The agent creates a frame and transfers the control to the hypervisor. The hypervisor decrypts the instructions, and interprets them until an uninterpretable instruction is reached. Then the hypervisor erases all the other instructions and returns control to the JVM TI agent, which interprets the instruction and either transfers the control to the hypervisor or returns control back to the JVM.

The interpreter constructs a frame, a data structure which constitutes the execution environment of the current method invocation (including the encrypted bytecodes of the current method), and transfers control to the hypervisor. The hypervisor decrypts the bytecodes and starts interpreting them one-by-one until it reaches an opcode which requires cooperation with the JVM. At this point, the hypervisor returns control to the agent and provides it with the instruction, which it could not interpret, in decrypted form. The agent proceeds by interpreting the instruction using JVM TI and JNI and then transfers control back to the hypervisor. Figure 6 presents the control flow diagram of the system operation.

7 Co-Interpretation

The interpretation is performed by two interpreters: one is embedded in the JVM TI agent and the other is embedded in the hypervisor (further reference to the thin-hypervisor will be simply: ”hypervisor”). Each opcode is interpreted by only one of the two interpreters. When one interpreter cannot continue interpretation, it transfers the control to the other interpreter. The interpreters share a data structure, which we call a frame, and in which they store the intermediate results of the interpretation as well as some additional information. The Frame’s structure is depicted in Figure 7 and explained below.

We want to enable the interpreter, which is embedded in the hypervisor, to interpret as many instructions as possible. Many instructions operate only on the stack and the program counter (PC). These instructions include the following

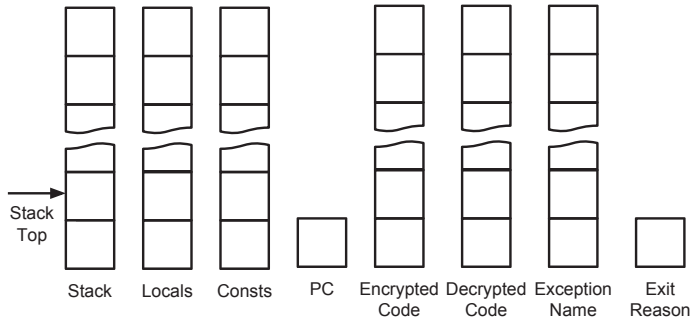

Fig. 7. Frame's structure. The frame contains the computation stack and a pointer to its top element. The frame includes copies of all the local variables and the constant pool. The program counter contains the location of the next instruction to be executed. The "encrypted code" buffer contains the encrypted bytecode of the current method, which is then decrypted and interpreted by the hypervisor. The "decrypted code" buffer contains the last instruction that could not be interpreted by the hypervisor. If the hypervisor encountered an abnormal condition, it reports its nature through the exception name buffer and sets the "exit reason" field accordingly.

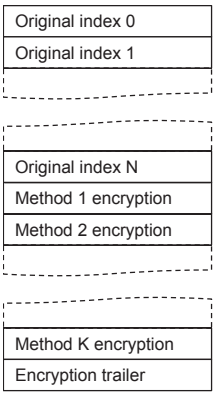

Fig. 4. Extended constant pool.

groups of instructions: arithmetic/logic, type conversion, stack management, control transfer. These instructions require the following information to be included in the stack: (a) PC, (b) stack. The load and store group of instructions allow the pushing of the value of local variables and constants onto the stack. In order to enable the hypervisor to interpret instructions in these groups, we include the (c) constant pool and the (d) local variables in the frame.

Instructions that belong to the following groups are interpreted by the JVM TI agent: object creation and manipulation, method invocation and return, and others. These instructions require cooperation with the JVM. For example, the *getfield* instruction, which pushes onto the stack the value of the specified field in the specified object, must inspect the internal representation of the object as defined by the JVM. Another example is the *return* instruction, which terminates execution of the current method. This instruction must modify the internal representation of the stack trace, which is managed by the JVM. Therefore, all these instructions are interpreted by the JVM TI agent via JNI and JVM TI functions.

In addition to the data structures which are used during interpretation, the frame includes three data structures which are used for communication between the two interpreters: (a) encrypted code, (b) decrypted code, (c) exit reason. Before transferring the control to the hypervisor, the encrypted code buffer is filled with the encrypted bytecodes of the current method by the JVM TI agent . The hypervisor decrypts the buffer and begins interpretation until it reaches an instruction, which cannot be interpreted (inside the hypervisor). This instruction is written to the decrypted code buffer and the exit reason is set to signify that the interpretation was suspended due to an uninterpretable instruction. The JVM TI agent interprets this single instruction and the process continues.

Exceptions are an essential part of Java; they are embedded into the low level bytecode instructions. Our interpreter supports exceptions (actually the support for asynchronous exceptions is partial) both in instructions that are interpreted by the JVM TI agent and those that are interpreted by the hypervisor. Clearly, our interpreter must cooperate with the JVM since it is possible that an encrypted method throws an exception which is handled by a non-encrypted method and vice-versa.

The implementation of exceptions in our interpreter can be divided into two parts: exception generation and exception handling. We begin our discussion with exception generation. The interpreter should generate an exception when it executes an instruction which generates an exception either explicitly (by executing the *athrow* instruction) or implicitly (e.g. by executing the *idiv* instruction with invalid arguments). The JVM TI agent delivers an exception to the JVM by calling the Throw or ThrowNew functions of JNI. The former function delivers the specified object as an exception. The latter function allocates a new object of the specified class and then delivers this newly allocated object as an exception.

Unfortunately, the hypervisor cannot call JNI functions directly. Therefore, whenever the hypervisor detects an abnormal condition, it transfers the control to the JVM TI agent. The nature of the exception is delivered through the

invokevirtual	8.9	invokeinterface	1.1	iaload	0.3
getfield	5.4	iastore	1.1	newarray	0.2
invokespecial	3.8	ireturn	1	instanceof	0.2
new	2.5	checkcast	0.7	ifacmpne	0.2
putfield	2.1	bastore	0.6	sastore	0.1
invokestatic	1.7	athrow	0.6	monitorexit	0.1
return	1.6	putstatic	0.4	lastore	0.1
getstatic	1.6	anewarray	0.4	castore	0.1
areturn	1.3	aaload	0.4	baload	0.1
aastore	1.2	arraylength	0.3	**Total:**	**38.1**

TABLE 1
Frequencies of uninterpretable instructions as reported by [38]

exception field of the current frame. The JVM TI agent then delivers the exception on the hypervisor's behalf.

In order to locate the correct exception handler, the JVM traverses the call stack of the currently executing methods. For each method, the JVM inspects the location in which the execution of the method was suspended and transferred to another method. These locations are part of the internal state of the JVM. The JVM updates these locations during program execution.

Our interpreters, however, cannot modify these locations directly, leaving the locations at 0 in all the encrypted methods. Whenever our interpreters need to modify the location of the currently executing method, they install a breakpoint at a desired location and return control to the JVM. The JVM executes the instrumented bytecode of the method (generally NOPS and a jump to the beginning at the end), as described in section 6, until it reaches the installed breakpoint, and transfers the control back to the JVM TI agent, which continues the interpretation process. Since our interpreter can affect only the location of the currently executing method, it must update the location before calling other methods.

The interpreter handles exceptions by intercepting the ExceptionHandled event of JVM TI. This event occurs when the JVM resolves an exception handler for a thrown exception. The interception function receives the exception object and the location of the exception handler. If the exception handler is not encrypted, the control is returned to the JVM. Otherwise, the interpreter pushes the exception object onto the stack and begins the interpretation process from the specified location.

8 Measurements

According to [38], invokevirtual is the second most popular instruction (appears with 8.9% frequency) and getfield is the fourth most popular instruction (5.4%). Unfortunately, these instruction cannot be interpreted inside the hypervisor, and, therefore, they are delivered in a decrypted form to the JVM TI agent, which is not considered secure. According to the statistics in Table 1, the hypervisor delivers about 38% of the instructions in a decrypted form back to the JVM TI. Therefore, in practice, only about 60% of the instructions in an encrypted class are actually hidden from an adversary.

8.1 Performance Benchmarks

To compare performance of protected-Java vs. non-protected-Java, two empirical measurements were conducted.

Algorithm 1 Algorithm measuring the correlation between instruction sequence length and its execution time.

```
for i = 1,10000 do
    t = System.nanoTime()
    baseTime += System.nanoTime() - t
    baseTime /= 10000
end for
for all k ∈ {10, 20, 30, ..., 190, 200, 400, 600, ..., 10000} do
    for i = 1,10000 do
        t = System.nanoTime()
        for i = 1,k do
            nothing
        end for
        sumTime += System.nanoTime() - t
        sumTime /= 10000
    end for
end for
Output: baseTime, sumTime
```

8.1.1 Code Execution Throughput

The purpose of this study was to compare code interpretation of decrypted Java bytecode in the hypervisor to regular, non-protected, JVM code interpretation. Algorithm 1 presents the pseudo code of a method that was run in protected and unprotected mode.

The first repeat block measures the overhead associated with system time measurement during 10000 iterations. The second repeat block performs the actual timed measurement. The number of interpreted instructions measured are a function of parameter k. The Java bytecodes measured are an empty *for* loop and include the instructions to manipulate the control variable and to cycle the loop. The value of k governs the number of instructions processed during each iteration. Measurements are performed for k assuming values 10 to 200 at increments of 10 and then 200 to 1000 — at increments of 200. To cancel out random measurement errors as a result of asynchronous events, 10000 iterations are performed for each value of k and the average value is output. The difference between *sumTime* and *baseTime*, the overhead measurement, is the net time duration of the instruction interpretation process. When measuring protected Java interpretation the *baseTime* of the non-protected java is subtracted from the measured result, in order to include in the time measurement a hypervisor exit and a hypervisor entry from/to the agent which is performed in order to call the System.nanoTime() method.

8.1.2 Results

```
Non-protected Java baseTime : 55 nanoseconds
Protected Java baseTime: 28726 nanoseconds
```

Since the System.nanoTime() method call is always executed by the JVM (it is not protected), the overhead time difference (28671 nanoseconds) is attributed to the transitions between the hypervisor and the agent.

The measurement result for the span of k values is plotted in Figure 8 on a logarithmic scale. Note that the transition overhead is significant (as compared to code interpretation) up until $k = 1000$. Since the size of the empty *for* loop in bytecode is about 10 bytes, it can be determined

Algorithm 2 Algorithm measuring the correlation between function's number of arguments and its invocation time.

```
baseTime = System.nanoTime()
for i = 1,40000 do
    nothing
end for
baseTime = System.nanoTime() – baseTime
for all k=0,15 do
    T_k = System.nanoTime()
    for i = 1,40000 do
        Call f_k(0, 1, ..., k)
    end for
    T_k = System.nanoTime() – T_k
    T_k /= 40000
end for
Output: baseTime, T_0, T_1, ..., T_15
```

that the transition overhead is significant for bytecode sizes of up to 10000 bytes. For larger bytecodes the performance comparison is stable at about a 1:10 factor.

While interpretation performance in the hypervisor can be optimized to achieve a result better than 1:10, this measurement shows that for all practical purposes the transition time will overshadow this. Therefore, optimization efforts should be concentrated there.

8.1.3 Protected Method Invocation

The purpose of this study was to measure the overhead of calling a protected method as a function of the number of its parameters. When a protected method is called, the agent needs to construct the frame context and transfer control to the hypervisor (via a hypercall). The hypervisor needs to locate the encrypted bytecode, decrypt it and perform the local interpretation. When complete, it needs to return control to the agent, which will adjust the JVM frame context.

To measure this entire process, functions of a varying number of parameters were called and the call procedure was timed. The function contents were identical:

$$f_k(\text{int } p_0, \text{int } p_1, ..., \text{int } p_k)\ \{p_0 = p_0 * p_0;\}$$

As in the previous study, each measurement was carried out multiple times (400000) to reduce error, and a *baseTime* was calculated to reflect the overhead associated with managing the loop process, as shown in Algorithm 2.

The *baseTime* was subtracted from the function call measurements results. Three types of measurements were conducted, each for all values of k:

- Non-protected Java run
- Calling method protected, callees non-protected
- Calling method non-protected, callees protected

Figure 9 plots these measurements on a logarithmic scale.

8.1.4 Results

The largest overhead was acquired when the callees were encrypted, since this operation is the most involved: requiring preparation of the environment, transferring to the hypervisor, decrypting, interpreting in hypervisor and restoring the environment. It is roughly 1.5 to 2 orders of magnitude greater as compared to the case where only the caller was protected, since this has moderate overhead, only requiring the hypervisor to prepare the environment and transfer control to the agent.

The number of parameters generally increase the timing linearly, as can be expected. However, when comparing the two protected cases, it can be seen that the gap between the results increases with the number of parameters. This indicates that the overhead of preparing and restoring the environment is more significant when the callee function is protected.

9 Future Work

The largest rival for Java in the world of managed languages is the .NET framework which was published by Microsoft on February 11, 2002. The framework itself is based on a language called "Common Intermediate Language" (CIL), formerly known as "Microsoft Intermediate Language" (MSIL) and a "Common Language Runtime" (CLR), whose responsibilities are to execute a given file written in CIL [39]. In order for a given programming language to be a .NET language it only needs a compiler from that language into CIL code. When Microsoft released the .NET framework, it also released the C# language as an abstraction for CIL along with the C# CIL compiler. As of 2015, it is the primary language for the .NET framework, although other languages exist (e.g., J#, F#, VB.NET and many more). Some of them were created by Microsoft, and some by other developers and companies. In this section we shall discuss the basic mechanisms of the CLR, the differences from JVM and why the solution we described in this paper does not apply to the .NET framework.

When a developer compiles a program written in some .NET language, e.g., C#, it actually compiles into an executable containing a CIL source code semantically identical to the original C# source code. Once the executable file is executed it loads the CLR into it and that CLR takes over and executes the program. It does so by means of a Just In Time (JIT) compiler — that is, whenever a piece of code is called for the first time it compiles into native machine code and only then it runs. This is in contrast to Java where the JVM interprets the piece of code each and every time it's being executed.

Therefore in order to come up with a solution for CIL source code, we need to accommodate the fact that the executable is actually compiled by a third party (the CLR) into native machine code and is then executed on its own (but governed by that CLR). Another design issue with .NET we have to take to consideration is that a managed exception cannot traverse the boundaries of the CLR — that is, exception cannot move from managed to unmanaged world outside the CLR. The current solution for Java uses an agent attached to the JVM, which intercepts the method invocation event. When the event occurs, the agent simulates execution of the invoked method. Since that agent has full access to the program memory, it can inspect and manipulate objects and primitive types along the method's simulation. The counterpart for that agent in the world of .NET are a set

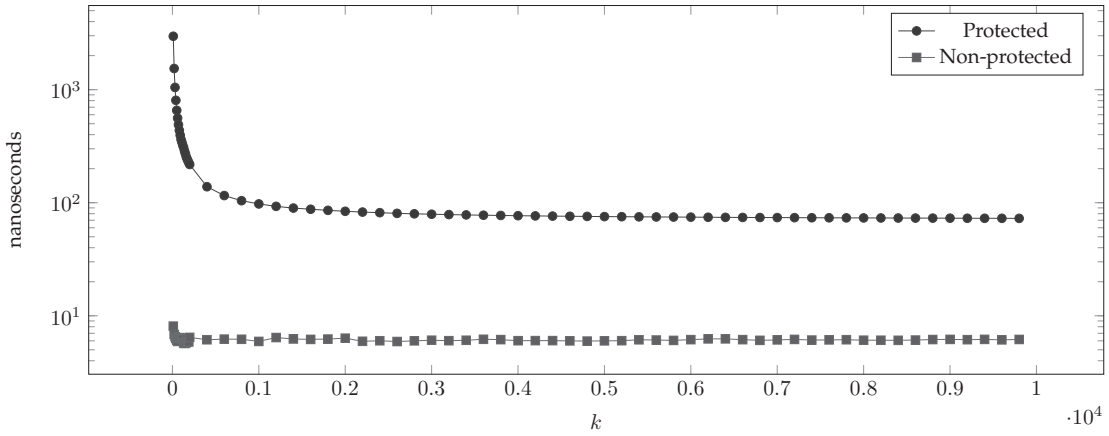

Fig. 8. Execution time of Algorithm 1 in nanoseconds. The figure presents two graphs that correspond to two execution modes: (1) protected mode, in which the algorithm is realized by an encrypted method, (2) non-protected mode, in which the algorithm is realized by a regular, non-encrypted method.

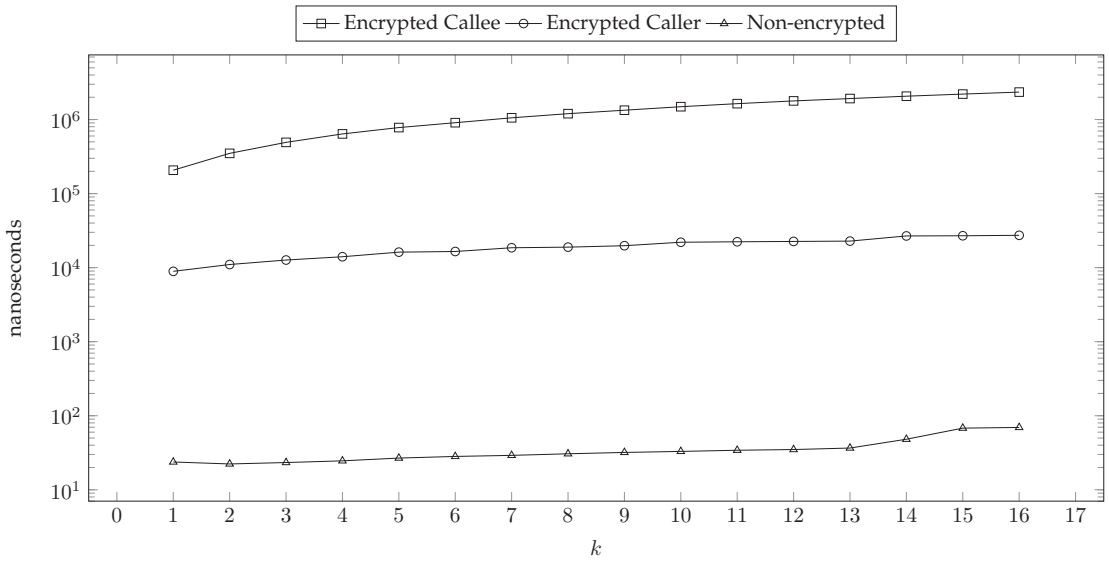

Fig. 9. Execution time of Algorithm 2 in nanoseconds. The figure presents three graphs that correspond to three execution modes: (1) the callee, i.e. the function f_k, is encrypted (2) the caller is encrypted (3) neither the caller nor the callee are encrypted.

of APIs called ".NET Unmanaged API" [40]. Within them, the following APIs are of interest for us:

1) CLR Hosting API - Intended to allow any native application — that is, a C/C++ application — to host the CLR in its memory and allows us to gain access to advanced APIs (which are described below), and on top of that it allows us to host and execute a .NET application directly from that native application.
2) Profiling API - Intended for program logging and monitoring, this API enables a developer to create a program called profiler which the CLR incorporates at certain extension points, such as object creation, function compilation and even function enter and leave. The profiler can even be used to rewrite the CIL source code of the function before it is compiled. The profiler enables program monitoring and memory inspection, but it cannot be used for memory manipulation, at least not in a straightforward way, since in order to do so it must interact with the CLR which is not supported by the API itself.
3) Reflection API - Intended to allow inspection and execution of .NET objects and functions from a native application. Unlike the profiling API, this one does not grant access to the program CIL source

code; gaining access to this API requires direct access to the CLR.

4) Debugging API - Intended to allow any developer to write a debugger for .NET applications. This API can access any aspect of the running program, including its memory, parameters (static and non-static), and CIL source code. A program utilizing this API will have the debugger running on a separate thread than the .NET application itself.

Using the above APIs, several applications can be imagined which mimic the protection scheme devised in the above paper, but they all fail due to lack of support from the APIs themselves. All of these pitfalls are exception related, and since exceptions are a big part of the language, and developers depend on them, this pitfalls cross out use of these APIs.

First the debugging API will be examined, as it is the simplest to refute. The main pitfall with this API is that it is running on a separate thread than the rest of the application. Hypothetically, if an interpreter was created and identified a division by zero, it should throw an exception. However, since the interpreter is running on another (native) thread and the .NET program is running on a different managed thread, the debugger cannot inject an exception to the .NET program. Therefore, program flow (which relies on exception catching and handling) cannot continue as it should.

Next, the profiler API will be examined. Using this API, function enters can be caught, and subsequently run in interpreter mode within the profiler. However, a problem occurs whenever object creation is required — the profiler can read existing objects, but it cannot call function or create objects for that manner. Therefore, using it as an interpreter is useless.

Lastly, there is the combination of Profiler and Reflection. Before examining this option, it must be noted that using the CLR within the profiler is not supported and can cause unexpected results. Despite that, in exploring this option, a native application is created, and the CLR is imported into it using the hosting API. The Reflection API is imported using the CLR object created before and placed in a shared memory space for later use. The CLR object is used to start the managed application that we wish to interpret. Before continuing, it is important to note that both the native app and the CLR (and therefore the managed application and reflection object) all live in the same application, and can access the memory of the other (which would not be possible through Microsoft). Once the profiler catches a function enter event, the CIL source code is accessed and the interpretation begins. Whenever we need to access a function or an object we use the reflection object in the shared memory to do so. The problem with this option arises, once again, from exceptions. When an exception needs to be raised from the interpreter, or even from a function called by the interpreter, the exception has to propagate through the native code. As stated before, this cannot be done — the exception object cannot traverse the boundaries of the managed code. Therefore, when the exception leaves the profiler and is caught in the managed code, only an SEH exception is identified — that is, a native exception which cannot be used in the same manner as managed exceptions. This, of course, affects the flow of a program which is built on exception handling. Therefore, this solution is also deemed unacceptable.

As proven with previous examples, the main problem which occurs with interpreting .NET applications is exception handling. Since exceptions cannot propagate correctly from unmanaged to managed code, we cannot use an application which lives outside of the CLR to interpret the application, at least not with the current toolset Microsoft provides with its Unmanaged APIs. Therefore, protecting .NET managed code requires direct CLR manipulation which will be described in future work.

10 Conclusions

As has been shown, Java programs can be, at least partially, protected from an adversary. We believe that this degree of protection is sufficient in cases where traditionally obfuscation was used. In other cases, which require a higher degree of protection, we suggest either avoiding using uninterpretable (by the hypervisor) instructions or use a tool which can reduce the frequency of uninterpretable instruction (by inlining methods, for instance).

While the performance penalty can be significant (2-2.5 orders of magnitude) on frequent transitions between the hypervisor and the JVM TI agent, the performance improves when longer sequences of instructions can be interpreted in the hypervisor at once. Obviously, sporadic execution of the protected parts of a Java program has little effect on the overall performance of the program. In case of repeated invocation of the protected parts, we suggest restraining the use of uninterpretable instructions, such as *getfield*, *setfield* and *invokevirtual*, in those parts. To conclude, by following a set of simple guidelines and using the proposed method, a developer can obtain a secure and reasonably efficient Java program.

References

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

[2] 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.

[3] *Themida*, http://www.oreans.com/, Oreans.

[4] *VMProtect*, http://vmpsoft.com/, VMProtect Software.

[5] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855876.1855877

[6] L. Bohne, "Pandora's Bochs: Automated Unpacking of Malware," 2008.

[7] D. Schellekens, B. Wyseur, and B. Preneel, "Remote Attestation on Legacy Operating Systems with Trusted Platform Modules," *Sci. Comput. Program.*, vol. 74, no. 1-2, pp. 13–22, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2008.09.005

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

[9] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A Trusted Open Platform," *Computer*, vol. 36, no. 7, pp. 55–62, Jul. 2003. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1212691

[10] C. Tarnovsky, "Semiconductor Security Awareness Today and yesterday," in *Blackhat*, 2010. [Online]. Available: https://www.youtube.com/watch?v=WXX00tRKOlw

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

[12] D. P. Delorey, C. D. Knutson, and C. Giraud-Carrier, "Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects," in *Second International Workshop on Public Data about Software Development (WoPDaSD'07)*, 2007.

[13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification*, Oracle Corporation.

[14] "The Java Language Specification." Oracle Corporation, 2015.

[15] A. Chander, J. C. Mitchell, and I. Shin, "Mobile code security by Java bytecode instrumentation," in *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, vol. 2. IEEE, 2001, pp. 27–40.

[16] H. B. Lee and B. G. Zorn, "BIT: A Tool for Instrumenting Java Bytecodes," in *USENIX Symposium on Internet technologies and Systems*, 1997, pp. 73–82.

[17] M. Harkema, D. Quartel, B. Gijsen, and R. D. van der Mei, "Performance monitoring of Java applications," in *Proceedings of the 3rd international workshop on Software and performance*. ACM, 2002, pp. 114–127.

[18] *Java Virtual Machine Tool Interface*, http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html, Oracle Corporation.

[19] W. Binder and J. Hulaas, "Exact and portable profiling for the jvm using bytecode instruction counting," *Electronic Notes in Theoretical Computer Science*, vol. 164, no. 3, pp. 45–64, 2006.

[20] J. Howarth, I. Altas, and B. Dalgarno, "Information Flow Control Using the Java Virtual Machine Tool Interface (JVMTI)," in *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*. IEEE, 2010, pp. 689–695.

[21] F. W. Long, "Software vulnerabilities in Java," 2005.

[22] M. Luedde, "Low impact debugging protocol," Nov. 13 2012, US Patent 8,312,438.

[23] 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. [Online]. Available: http://doi.acm.org/10.1145/361011.361073

[24] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[25] R. J. Creasy, "The Origin of the VM/370 Time-sharing System," *IBM J. Res. Dev.*, vol. 25, no. 5, pp. 483–490, Sep. 1981. [Online]. Available: http://dx.doi.org/10.1147/rd.255.0483

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

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

[28] 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. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[29] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The Turtles Project: Design and Implementation of Nested Virtualization," in *OSDI*, vol. 10, 2010, pp. 423–436.

[30] J. Rutkowska and A. Tereshkin, "IsGameOver(), anyone?" in *Blackhat 2007*, 2007. [Online]. Available: http://invisiblethingslab.com/resources/bh07/IsGameOver.pdf

[31] ——, "Bluepilling the xen hypervisor," *Black Hat USA*, 2008.

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

[33] 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. [Online]. Available: http://doi.acm.org/10.1145/1508293.1508311

[34] 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. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774131

[35] R. Kennell and L. H. Jamieson, "Establishing the Genuinity of Remote Computer Systems," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251374

[36] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095812

[37] 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.

[38] C. Collberg, G. Myles, and M. Stepp, "An Empirical Study of Java Bytecode Programs," *Softw. Pract. Exper.*, vol. 37, no. 6, pp. 581–641, May 2007. [Online]. Available: http://dx.doi.org/10.1002/spe.v37:6

[39] "Standard ECMA-335: Common Language Infrastructure (CLI)." ECMA International, 2012.

[40] "Unmanaged API Reference." Microsoft Corporation.

Michael Kiperberg was born in Ukraine in 1987 and migrated to Israel in 1997. He received his B.Sc. and M.Sc. (Cum Laude) in Computer Science from Tel Aviv University, Israel in 2012. Michael is currently a Ph.D. student at the University of Jyvaskyla, under the supervision of Prof. Pekka Neittaanmaki. Michael served as an academic officer in the Israeli Air Force, and now he is a Chief Scientist in a startup company.

Amit Resh was born in Haifa, Israel, in 1959. He received his B.Sc. in Computer-Engineering and MBA from the Technion, Israel Institute of Technology, in 1986 and 2001 respectively. In 2013 he received his M.Sc. from the University of Jyvaskyla, Finland. He has more than 25 years of professional experience in hi-tech companies in Israel and the USA. He has previously worked as Program-Manager at Apple and as VP of R&D at Connect One, as well as other companies in the embedded-systems industry. Currently he is COO of TrulyProtect, a startup company developing trusted computing systems based on virtualization technology. As of 2014 he is working towards his Ph.D. at the University of Jyvaskyla, Finland.

Asaf Algawi was born in Haifa, Israel in 1986. He received his B.Sc. in Information Systems Engineering from Ben-Gurion University of the Negev, Israel in 2009. He has 6 years of professional experience in hi-tech military units in the IDF, these include 3 years as a system analyst and another 3 years leading a small team of .NET developers. As of 2014 he is working towards his M.Sc. at the University of Jyvaskyla, Finland.

Nezer Jacob Zaidenberg was born in Tel Aviv, Israel, in 1979. He received his B.Sc in Computer Science and Statistics and Operations Research in 1999, M.Sc in Operations Research in 2001, and MBA in 2006 from Tel Aviv University. Nezer completed his Ph.D. studies in 2012 at the University of Jyvaskyla. Nezer is the owner of Warp speed solution consulting group and has been consulting to NDS, IBM, Kardan VC, Sintecmedia, Videocells among others. Currently Nezer is CTO of TrulyProtect.

PVII

SYSTEM FOR EXECUTING ENCRYPTED NATIVE PROGRAMS

by

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

IEEE Symposium on Security and Privacy

System for Executing Encrypted Native Programs

Michael Kiperberg*, Amit Resh*, Roee Leon*, Nezer J. Zaidenberg*

*Department of Mathematical Information Technology, University of Jyvaskyla, Finland
{michael, amit, roee, nezer}@trulyprotect.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, it is proven 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 a 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 the just-in-time decryption: in-place and buffered execution. The former being safer, while the latter boasts better performance**

***Index Terms*—Trusted Computing, Hypervisor, Virtualization, Remote Attestation**

I. 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 to understand by deliberately cluttering the code with useless, confusing pieces of additional software syntax or instructions. However, even when changing the 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]–[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.

The paper is organized as follows. Section II presents the structure of executable files, and describes the transformation applied to these files by the encryption tool. Hypervisors and their role in security are discussed in section III. Section IV outlines the attestation protocol, which is performed during hypervisor's initialization. We discuss the execution process of an encrypted function in section V. The performance and security measurements of the system are presented in section

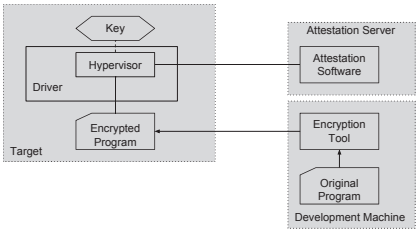

Fig. 1. Native code protection system. The original program is encrypted before its distribution. The encryption key is stored in the attestation server, which delivers it to the hypervisor in the target machine upon successful attestation. The hypervisor is initialized by a driver, which also hosts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

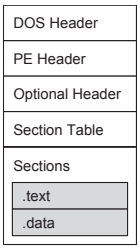

Fig. 2. Structure of a Windows PE file. The structure contains a variable number of sections. Two of the most common sections are presented.

VI. Section VII discusses the limitations of the presented system and its possible extensions. Section VIII summarizes the results of this paper.

II. 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.

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.

As will be explained in section V, 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

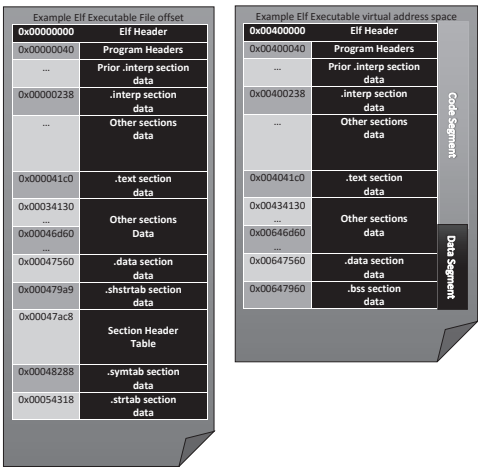

Fig. 3. The left image represents the structure of an ELF file as it is stored in disk. The right image represents the structure of an ELF file as it is loaded to memory.

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 transformation.

III. 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.

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 or the cryptographic key.

Once this environment is correctly configured, a thin hy-

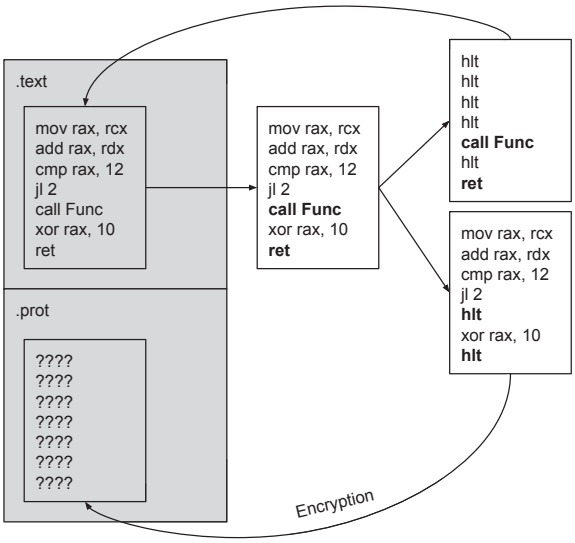

Fig. 4. Example of an encryption process of a single function. The encryption begins by classifying instruction is encryptable (normal face) and non-encryptable (bold face), and creating to copies. The complementary instructions in each copy are replaced by halts. Finally, one copy is written over the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

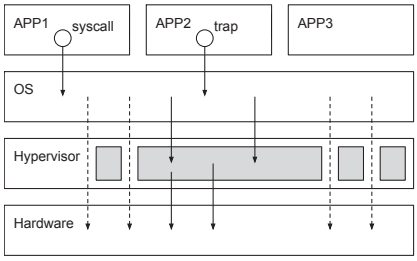

Fig. 5. Thin hypervisor. The hypervisor runs in a higher privilege level than the operating system. System calls, traps, exceptions, and other interrupts, transfer control from user mode applications to the operating system. The operating system handles these conditions by requesting some service from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

pervisor 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.

IV. Remote Attestation

The problem of remote software authentication, determining whether a remote computer system is running the correct version of a software, is well known [5], [19]–[25]. Equipped with a remote authentication method, a service provider can prevent an unauthenticated remote software from obtaining some secret information or some privileged service. For example, only authenticated gaming consoles can be allowed to connect to the gaming networks [26]–[28], and only authenticated bank terminals can be allowed to fetch records from the bank database [29].

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

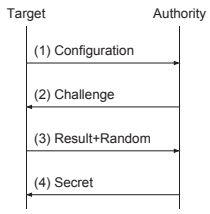

Fig. 6. The attestation protocol between the authentication authority and the target machine. The protocol consists of four messages. The first two messages are sent unencrypted, while the two last messages are encrypted. The third message is encrypted by the public key of the authentication authority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

frame. The underlying assumption, which is shared by all such authentication methods, is that only an authentic system can compute the correct result within the predefined time-frame. The methods differ in the means by which (and if) they satisfy this underlying assumption.

Kennell and Jamieson proposed [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 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 target's authenticity. The authentication authority encrypts the secret information with a random value obtained from message (3) acting as the encryption key, and transmits the encrypted message to the target machine.

V. 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 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*.

A. 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 III. The hypervisor switches to the hot state when the memory contains sensitive information, which cannot be protected by a regular memory protection technique (using EPT), since its physical location is not known (or not constant). This switch 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 instruction, but the same is true when the software breakpoint instruction 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 III, the hypervisor intercepts general protection faults.

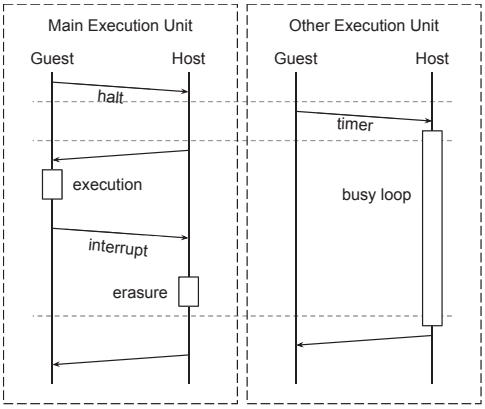

Fig. 7. Example of encrypted function execution. The figure depicts two execution units, each with two alternating states: guest and host.The dashed horizontal lines are synchronization barriers, i.e. everything above the line is guaranteed to complete before anything below the line starts.

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 behaviour 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.

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 Nth interrupt handler of the original IDT.

B. 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 III. 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. 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.

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

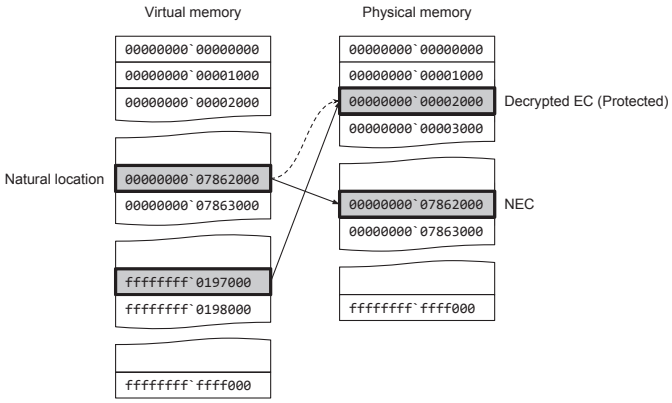

Fig. 8. Memory layout during buffered execution. The functions resided at virtual address 7862000, which is mapped to the physical address 7862000 (a coincidence). The encrypted code is decrypted to virtual address ffffffff`0197000 which is mapped to the physical address 2000. The hypervisor changes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

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.

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 behaviour of unknown code through its data.

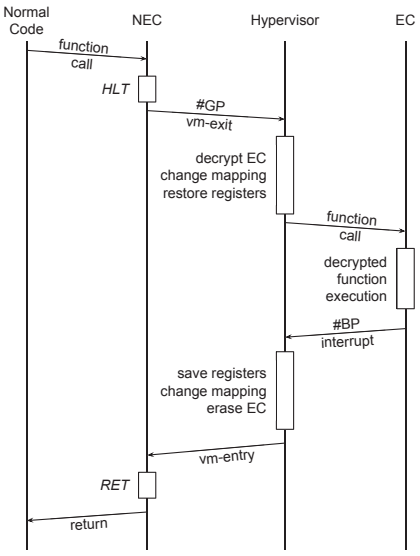

Fig. 9. Example of encrypted function execution in buffered execution mode. The figure depicts the control flow during the execution of an encrypted function.

VI. Measurements

This section presents a performance analysis of the two execution methods that were described in section V. The measurements were performed on programs with a single

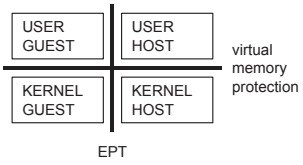

Fig. 10. Execution modes. The left column represents the guest mode, while the right column represents the host mode. The lower row represents the kernel mode, while the upper row represents the user mode. The host mode can protect itself from the guest mode through the EPT mechanism. The kernel mode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

encrypted function. The functions that were chosen do not call other functions. Therefore, these functions can be executed at once. The size of the functions varied, as will be explained below.

According to the in-place execution method, at the beginning of each execution cycle, the main execution unit freezes other execution units. This is usually accomplished with the following sequence of actions:

1) The main execution unit writes *freeze requests* to the data structures of other execution units.
2) The main execution unit triggers a vm-exit in other execution units.
3) Other execution units inspect their data structures.
4) Other execution units enter a busy-loop.
5) The main execution unit proceeds (to decryption, execution, etc.).

Usually, in order to perform some operation on a different execution unit, the current execution unit sends an inter-processor interrupt (IPI), which triggers an interrupt service routine. This is probably the most efficient, and at the same time, the most complicated solution. We chose to induce a vm-exit periodically on every execution unit, through the *preemption-timer* field of the VMCS, which defines the maximal amount of time that the guest can execute before returning to the hypervisor. Figure 11 depicts the relation between the execution time of a function and the preemption-timer interval. The figure also includes the execution times of the function under the following conditions:

- Regular, unencrypted, execution time.
- Execution time using the buffered method.
- Execution time using the in-place method, which skips the freeze step.

Figures 12 and 13 present the time division between the steps of the in-place and the buffered execution methods, respectively. The in-place execution method consists of the following steps:

- vm-exit
- execution units freezing
- decryption
- vm-entry
- instructions execution
- vm-exit
- erasure of the decrypted function
- execution units unfreezing
- vm-entry

Figure 12 presents the execution time of each step. The buffered execution method consists of the following steps:

- vm-exit
- virtual page table modification
- decryption
- instructions execution
- erasure of the decrypted function and virtual page table restoration
- vm-entry

Figure 13 presents the execution time of each step.

Finally, figure 14 compares the execution times of the in-place and the buffered execution methods for functions of different lengths.

VII. 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.

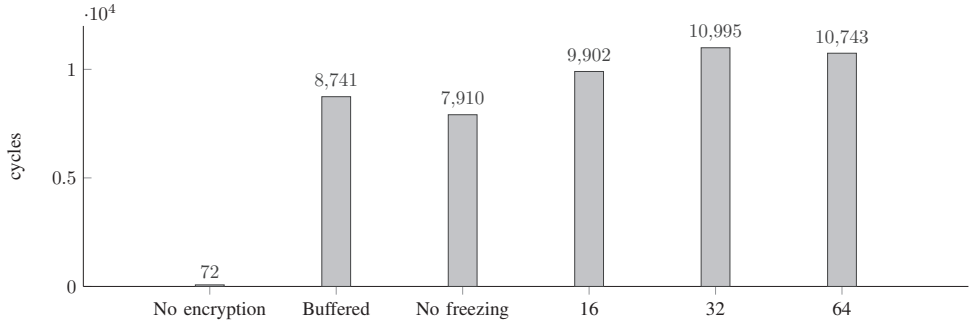

Fig. 11. Comparison of execution times as a function of the preemption-timer value. The columns represent the following cases (from left to right): regular execution — no encryption, buffered execution, in-place execution without freezing, in-place execution in which the preemption timer is set to 16, in-place execution in which the preemption timer is set to 32, in-place execution in which the preemption timer is set to 64.

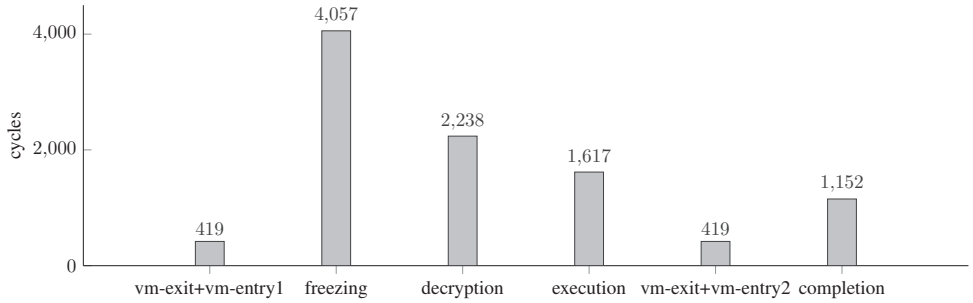

Fig. 12. Execution times of the steps in the in-place execution method (in which the preemption timer was set to 16). The columns represent the following steps (from left to right): entering and exiting the host; decrypting the function; executing the function; entering and exiting the host; erasing the decrypted function.

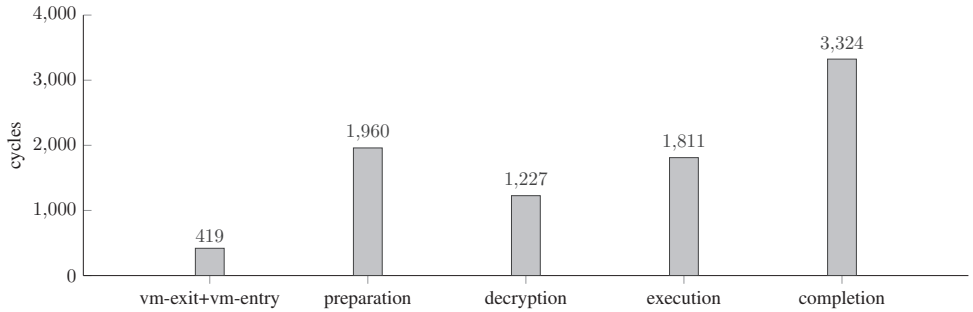

Fig. 13. Execution times of the buffered execution method steps. The columns represent the following steps (from left to right): entering and exiting the host; modifying the virtual page table and restoring the guest register; decrypting the function; executing the function; erasing the decrypted function, saving the registers, and restoring the virtual page table.

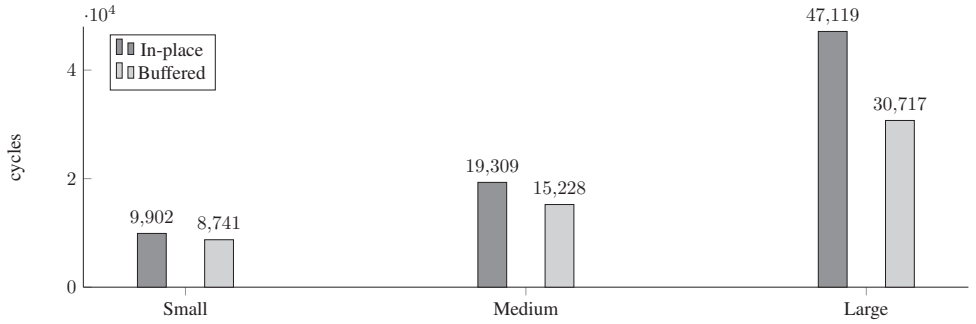

Fig. 14. Execution times of the in-place and the buffered execution methods for functions of different lengths. The small function is 92 bytes long. The medium function is 968 bytes long. The large function is 5056 bytes long.

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.

VIII. 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, penitentially 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. Measurements show that the larger overhead is 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 medium (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.

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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855876.1855877

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

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

[8] C. Tarnovsky, "Semiconductor Security Awareness Today and yesterday," in *Blackhat*, 2010. [Online]. Available: https://www.youtube.com/watch?v=WXX00tRKOlw

[9] ——, "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 64 and IA-32 Architectures Software Developer's Manual - Volume 3*, Intel Corporation, August 2007.

[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. [Online]. Available: http://doi.acm.org/10.1145/361011.361073

[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. [Online]. Available: http://doi.acm.org/10.1145/1508293.1508311

[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. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774131

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

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

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

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

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

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

[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/a_id/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.