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

1932-8184/$31.00 © 2013 IEEE

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.

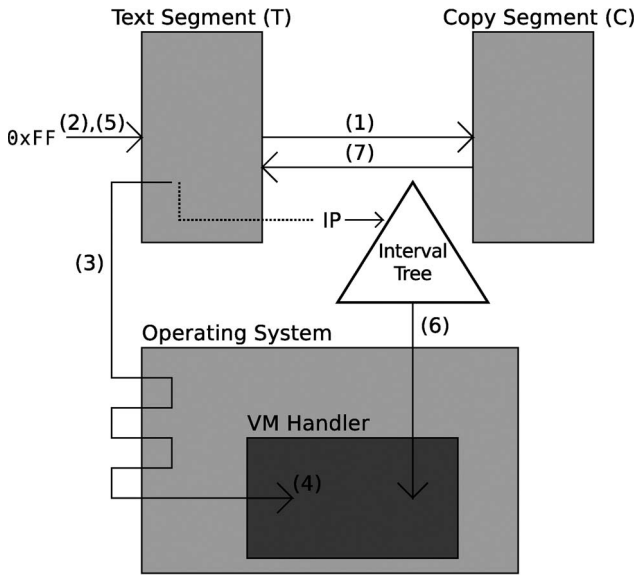


Fig. 1. Just-in-time decrypting.

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.

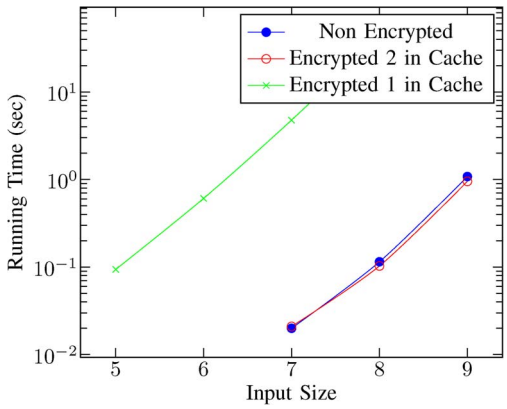


Fig. 2. Just-in-time decryption performance. Program running time (in seconds) as a function of input size. Note the logarithmic scale.

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

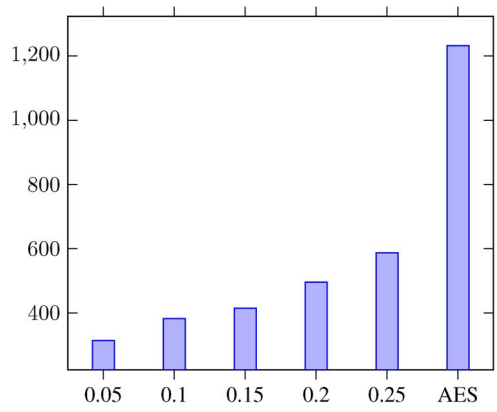


Fig. 3. Program decryption time (in cycles) using AES and our cipher with different values of p. $\ell = 4$. $\alpha = 14$.

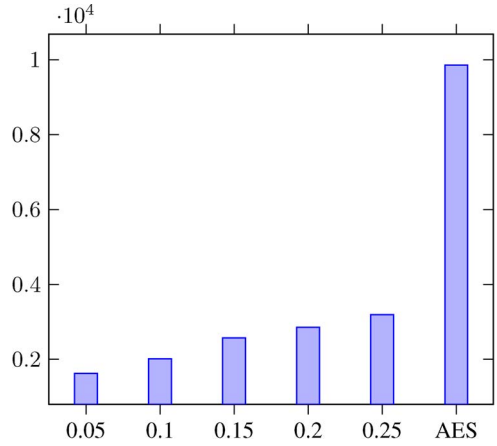


Fig. 4. Program decryption time (in cycles) using AES and our cipher with different values of p. $\ell = 4$. $\alpha = 112$.

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

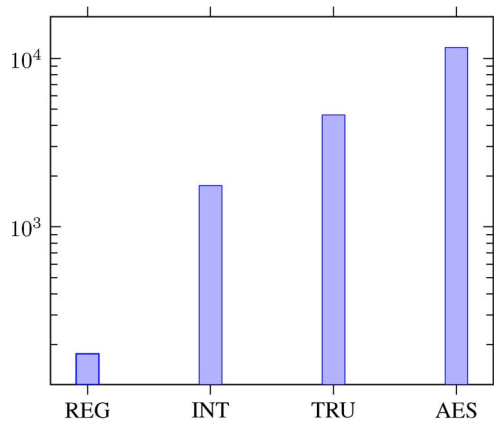


Fig. 5. Program decryption time (in cycles) using (left to right) a regular execution, interpretation, AES, and our cipher. $\ell = 4$. $\alpha = 112$. $p = 0.2$. Note the logarithmic scale.

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

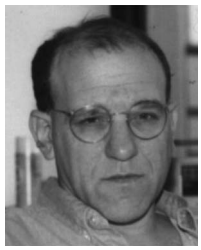
and audio, unless the content is also encrypted. Truly-Protect can be used to mask the decoding and decryption process.

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.