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 $p = 0.2$

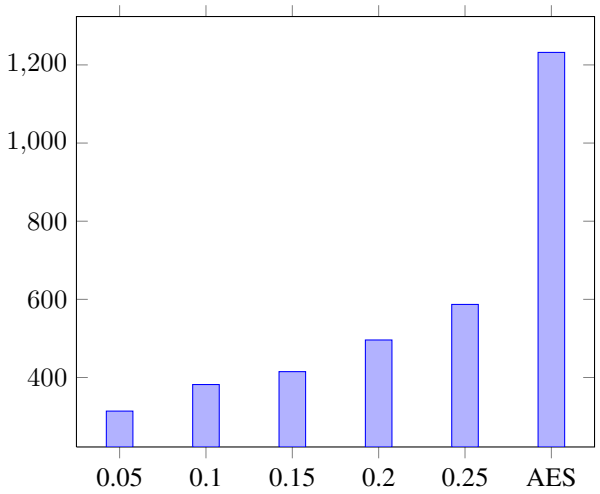

Fig. 2. Program decryption time (in cycles) using AES and our cipher with different values of p. $\ell = 4$. $\alpha = 14$.

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.

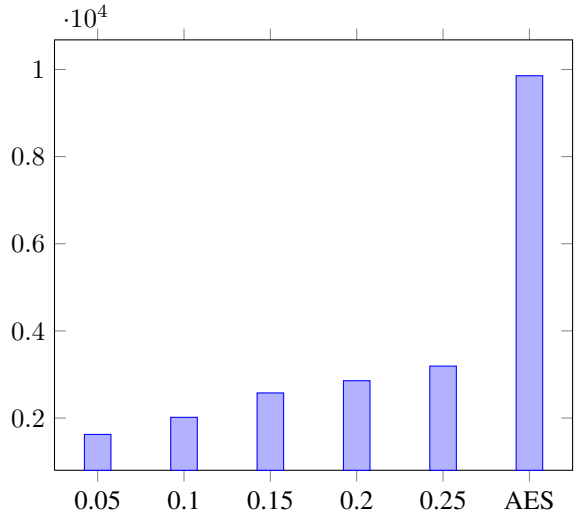

Fig. 3. Program decryption time (in cycles) using AES and our cipher with different values of p. $\ell = 4$. $\alpha = 112$.

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**