Program obfuscation by strong cryptography

Željko Vrba*†, Pål Halvorsen*†, Carsten Griwodz*†
*Simula Research Laboratory, Oslo, Norway
†Department of Informatics, University of Oslo, Norway
{zvrba,paalh,griff}@ifi.uio.no

***Abstract*—Program obfuscation is often employed by malware in order to avoid detection by anti-virus software, but it has many other legitimate uses, such as copy protection, software licensing or private computing in the cloud. In this paper, we present a program obfuscation method that is based on the combination of strong encryption of code and data and a CPU simulator (CSPIM) that implements the MIPS I instruction set. Our method is different from existing methods in that only a single word (32-bits) of the protected code or data is present as plain-text in main memory. Furthermore, our method allows the possibility of externally supplying the decryption key to the simulator. We have extensively tested the simulator, and it is able to successfully execute C programs compiled by the gcc cross-compiler. Even though purely software-based method cannot provide perfect protection, we argue that this approach significantly raises the bar for reverse-engineers, especially wen combined with existing program obfucation techniques.**

I. Introduction

Program obfuscation techniques aim to make the program's code unintelligible to a reverse-engineer (attacker). Such techniques are most often employed by computer viruses in order to avoid detection, but they have also many legitimate uses. For example, a software vendor may want to implement software licensing schemes that hinder unauthorized copying or use of their products. Another example are grid and cloud computing technologies [1], [2] offer enormous distributed computing resources at affordable price, but where different clients want to be protected from each other.

A common characteristic of both scenarios is that a deployed application is under full control of the administrator of the machine it is executing on, a situation that may be undesirable for several reasons. For example, the administrator can inspect the application's *code* to learn about its algorithms, some of which may be secret intellectual property, or maliciously manipulate the application's inputs and outputs. Likewise, the *data* processed by the application may be confidential, and its uncovering could damage the application's owner (and potentially bring financial gain to the administrator). Since hardware-assisted virtualization has become a standard feature of new Intel and AMD CPUs, monitoring (and potentially manipulation) of applications can be performed with minimal, almost undetectable, interference [3].

Using simulated CPUs for obfuscating code in order to protect it from reverse-engineering is not a new idea, and it is used by commercial tools like VMProtect [4] or Themida [5] as well as by virus-writers, who often use custom-designed instruction sets. The problem with custom instruction sets is lack of tools: there are no code generators from high-level languages, and developing one is a complex task. Alternatively, one could implement a translator from the target instruction set (e.g., i386 or AMD64) to the simulated instruction set. However, this is an extremely complex and tedious task for a CISC architecture such as AMD64, which has irregular instruction encoding and hundreds of instructions.

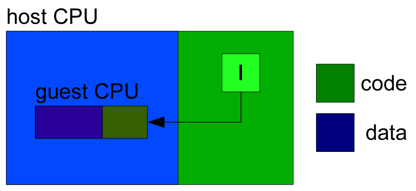

Fig. 1. Encrypted execution overview. The light-green rectangle labelled with "I" represents the portion of the host's code that interprets the guest code.

In this paper, we present an approach in which we combine strong encryption with a software-simulated CPU in order to hide the application's code and data from casual inspection and even from determined attackers. The high-level idea is shown in figure 1, where two main elements are the host and guest CPUs, each with its own code and data. The state of the guest CPU is part of the host CPU's *data*. The host CPU's *code* contains, alongside other functions, also the *interpreter* that is able to simulate the guest CPU and execute code written for it. The guest CPU's code and data are held *encrypted*; the interpreter decrypts them with fine granularity only when needed. Note also that the decryption key does not need to be a part of the host CPU's program or data – it may be, for example, acquired over the network, typed in by the user, or computed from environmental parameters, such as CPU model and amount of installed memory.

We have chosen to implement a simulator, which we have named CSPIM, for the basic MIPS I [6] instruction set, because this is one of the simplest CPUs for which the freely available gcc compiler can generate code. This means that the code to be protected can be developed in a high-level language (C or C++), and fully tested and debugged with the usual development tools. When the developer is convinced that the code is functioning correctly, he can cross-compile it for our simulator just before deployment, which saves time in the development cycle. Our approach is orthogonal to, and can be combined with, the existing code obfuscation techniques [7], [8], [9], which can be applied on both the host and guest executables.

The rest of this paper is organized as follows. In section II we discuss existing approaches to code and data obfuscation. In section III, we describe CSPIM design and implementation and discuss its limitations. In section IV, we describe our test applications and present some performance measurements. In section V, we present some envisioned usage scenarios and discuss security issues, and in section VII, we conclude and outline our envisioned future work.

II. Related work

There exist many MIPS I simulators, such as SPIM, gxemul, qemu, vmips and Nachos. All of them, except SPIM [10], implement full-system emulation, which has too large overhead for our purposes. Also, none of them satisfy the following two basic requirements:

- Availability of a clean library interface that allows the host application to control all aspects of simulation.
- Ability to run in a freestanding environment, i.e., that the simulator does not use any external library functions.

Because of the second requirement, the CPU simulation code is platform-independent and can be also easily interfaced with an OS kernel in order to provide further protection.

There exist executable code encryptors that use strong cryptography, such as Burneye [11] and Shiva [12], but their weakness is that they leave rather large portions of the executable's memory pages in plain-text. While they do employ some powerful anti-debugging techniques, such techniques prevent debugging only from user-space and can be defeated by using custom kernel modules or virtualization. Filiol [13] has outlined an architecture of a computer virus that is fully-encrypted and uses environmental data to derive its decryption keys. As with Shiva and Burneye, the virus is decrypted in its entirety before execution.

Similarly to Shiva, Morris and Nair [14] have proposed a secure context-switch mechanism, which is integrated with the OS's scheduler. When the scheduler is about to schedule a protected application, it decrypts all protected pages before it transfers control to it. When the application is about to be descheduled, all protected pages are reencrypted. Their approach works only on single-processor systems; in a multi-processor system, the attacker could stop the protected application (by sending it a SIGSTOP signal) just at the moment it is executing, and thus access its pages in plain-text. Furthermore, they limit the application's size to 4 MB, because encryption and decryption overheads become overwhelming beyond this limit. This method is easily defeated by program analysis techniques based on virtualization, such as Ether [3]. The hypervisor can stop the guest operating system just when it is about to perform context-switch and obtain the program's complete code and data in plain-text.

We know of only two other approaches that decrypt an instruction at a time. The first was used in the DOS virus called DarkParanoid [15], [16], which uses single-stepping to decrypt an instruction at a time. The "encryption" does not use a strong encryption algorithm, but a polymorphic engine which thwarts signature-based detection. While convenient, single-stepping invokes the CPU's exception handlers and thus incurs large run-time overheads, as we shall show later in this paper. The other solution [17] uses self-modifying code to execute encrypted i386 code. However, due to transparent interoperability with non-encrypted code and the complexity of disassembling i386 code, it does not support on-the-fly encryption and decryption of data accesses.

Flicker [18] uses the trusted platform module (TPM) and virtualization extensions to provide application secrecy and integrity. Flicker ensures that only unmodified applications are run, and it can, in addition, ensure that the application's data is never revealed to the rest of the operating system (and other applications) in plain-text. The encrypted data can additionally be protected from tampering. However, Flicker does not address *code* protection at all, and its current implementation has some serious limitations: it disables all interrupts and suspends the operating system as well as all processors but one.

In summary, we know of no published program protection methods that use strong encryption and leave only a tiny fraction (single machine word) of the protected program's code and data in plain-text. The method closest to ours is secure context switch [14], but it has two drawbacks: it is limited to single-processor machines, 2) even on a single-processor machine, it is easily defeated by using analysis toolkits based on virtualization, such as Ether [3].

III. The CSPIM simulator

CSPIM is implemented in portable C language[1] and has the following features:

- It simulates the integer instruction subset of MIPS I CPU in *little-endian* mode.
- The simulated CPU is fully jailed, i.e. the simulated program cannot by any means access the simulator's or the host program's state.
- Clean library interface that includes CPU control, instruction decoding, ELF loading,[2] symbol and address lookup.
- Extensive tests that test every instructon for correct operation.
- Implements most of SPIM [10] system calls with (almost) the same calling convention, and also adds some new system calls.
- RC5 encryption algorithm with 32-bit block size and 128-bit key for protecting code and data accesses.

The simulator has been tested on Solaris 10, Linux and Windows operating systems using native compilers (SunCC, gcc and VC, respectively) in 32- and 64-bit mode. In every configuration, the simulator has successfully executed the correctness tests, which have been compiled with the gcc cross-compiler.

In the next sections, we present in detail the simulator's internal structure and discuss its limitations.

A. Architecture

Figure 2 shows the internal organization of CSPIM. The host program uses the ELF component to prepare the MIPS

[1]The code is available at http://zvrba.net/software/cspim.html

[2]Executable and Linkable Format; commonly used on UNIX systems.

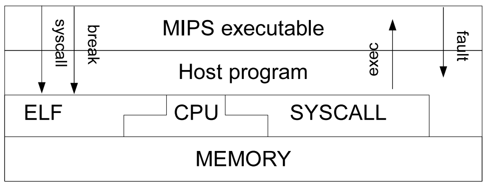

Fig. 2. The CSPIM architecture.

executable for execution, and the CPU component to actually run the guest program instruction by instruction. The outcome of each instruction may be success, fault, syscall, or break; the last two are generated by the respective MIPS instructions.

Memory layer. The memory layer handles all data transfers between the host and simulated CPUs. This is also the layer at which on-the-fly encryption and decryption of memory accesses takes place. The memory layer ensures that memory accesses cannot reach locations outside of the simulated CPU's memory space. If a simulated program attempts to access memory outside of the memory range allocated to the simulated CPU, the control is immediatelly returned to the host program with error code signaling invalid memory access. The memory layer also provides utility functions for copying data between guest and host CPUs.

CPU layer. The CPU layer maintains the state of the simulated CPU (e.g., registers) and decodes and executes single instructions. Memory reads and writes of all sizes (1, 2 and 4 bytes) are vectored through two function pointers, which makes it possible to plug-in various kinds of transformations, such as encryption and endian-swapping. The latter would enable CSPIM to run in little-endian mode also on big-endian platforms.

Figure 3 shows the address space in which the simulated program executes. `base` is the address in the *host's* address space where the memory allocated for the simulated CPU starts, and `memsz` is the total amount of memory allocated for the simulated CPU. Addresses generated by the *simulated* CPU are interpreted by the host as offsets into the memory region allocated to the simulated CPU. `_end` is the symbol that marks the end of the code, data and uninitialized data segments (i.e., it is the address where the heap starts), while `stksz` is the total amount of memory allocated for the stack. The heap grows upward and its current limit is known as "the break" (`brk`), while the stack grows downwards; `$sp` is the stack pointer register which points to the current stack top location.

ELF layer. The ELF layer prepares an in-memory ELF image for execution. This includes thorough validation of all segment and section data before they are used, copying text and data segments to their respective places in memory, and initializing the `$pc` (program counter) and `$gp` registers. The public API also provides functions for symbol lookup and address-to-symbol mapping. When compiling programs that target CSPIM, the following options must be given to the gcc compiler: `-mips1 -mno-check-zero-division -mlong-calls`; these options force the compiler to use only

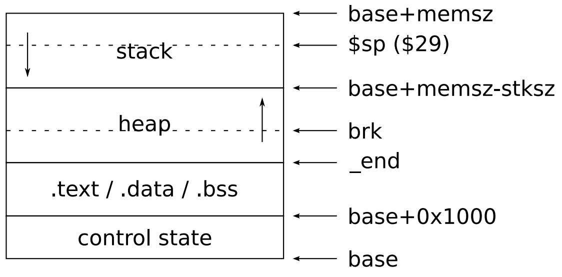

Fig. 3. The simulated program's memory layout. The dashed lines represent values that vary during execution.

the MIPS I instruction set and disable checks for division by 0.[3] Also, the following options must be given when linking object files to produce the final executable: `-Wl,-q -nostdlib -Ttext 0x1000`. These options instruct the linker to leave relocation data in the final ELF file, to not link in the standard libraries, and to set the address of the first instruction to `0x1000` (hexadecimal).

The ELF loader ensures that there is sufficient space for the program so that its segments do not overflow into the stack area. However, like the real CPU, the simulator does *not* check for stack overflows at run-time as this would prevent the simulated program from using the stack pointer as a temporary general-purpose register or from implementing coroutines. The simulator guards the first 4kB of the guest's address space from accesses by the guest program, which in effect provides NULL pointer protection. The host uses this area to store the simulator's control state.

System call layer. The system call layer is available only when the host application executes in a hosted environment, i.e., in user-space under an operating system. The system calls provide support for console I/O of characters, strings and integers, dynamic memory allocation through the `sbrk` call having the same semantics as under UNIX, and (binary) file I/O. A system call is invoked in three steps:

1) The system call parameters are placed in registers, following the usual MIPS calling convention.
2) The system call number is placed in register `$2`.
3) The `SYSCALL` instruction is invoked with the code field set to `0x9107C` (an arbitrary value).

This is almost the same system call convention as defined by the SPIM simulator. The only difference is that SPIM seems to ignore the code field of the `SYSCALL` instruction, whereas CSPIM mandates that it be set to the above value. The purpose of this check is to ensure that the calling code follows the SPIM calling convention; in the future, it could be also used to support different system call conventions.

To protect the host's file descriptors from the guest program, the guest CPU maintains its own file-descriptor (FD) table. FDs 0, 1 and 2 are always hard-wired to the host's `stdin`, `stdout` and `stderr` streams, while other guest's FDs are remapped to the host's. Thus, accessing a given FD in the

[3]Unlike many other CPUs, MIPS I does not generate an exception on division by 0; instead, it sets the result registers to undefined values.

guest will not necessarily access the same FD in the host. The maximum number of available file-descriptors in the guest is configurable at compile-time, and is by default set to 16.

Host: execution and error-handling. The host program is responsible for allocating the chunk of memory in which the guest will execute, loading the ELF image and initializing the guest CPU state, and controlling execution. The host controls execution of the guest CPU by repeatedly invoking a function that executes a *single instruction* at a time. This function returns an error code that indicates whether the instruction succeeded; possible failure causes are integer overflow, accessing an invalid address, attempting to execute an invalid instruction, encountering a `SYSCALL` or `BREAK` instruction, or because the simulator has detected an unrecoverable internal error.

Since the guest CPU cannot access host's memory, the host is also responsible for providing the guest program with inputs and getting the results out. CSPIM provides utility functions to move data between host and guest memory. Host can use the provided ELF symbol lookup functions to access data items in the guest's memory through symbolic names instead through absolute addresses, which are susceptible to change each time the guest program is recompiled.

B. Limitations

The current version of the simulator also departs in some ways from the real MIPS I CPU:

- Big-endian mode is not supported, neither in the host nor in the guest. However, support for big-endian hosts can be easily added by providing the memory layer with endianness-swapping functions.
- Coprocessor instruction sets is not supported. Specifically, neither floating point nor system control coprocessors are implemented, nor any kind of cache or virtual memory simulation. Floating-point support may be added in the future, but the other features are simply not necessary for correct execution of user-mode programs.
- Consequently, there is no way to handle exceptions from MIPS code – all exceptions are instead reflected to the host program for processing, as explained above. If an instruction in the branch delay slot causes an exception, the exception is reported to the host for that instruction.
- Delayed loads are not simulated; the target register is instead immediately loaded with memory contents. This departure from the specification does not matter for any correct program, and we assume that GCC generates correct code in this respect.[4]
- As with the real CPU, division by zero produces an undefined result – the current implementation always returns -1 for both quotient and remainder.

Despite these differences, CSPIM successfully and correctly executes programs compiled and linked by the mipsel-elf-gcc compiler, when using the previously given flags.

[4]This is not an issue even for incorrect programs, since the real CPU returns an *undefined* value if the program attempts to use the target register's value immediatelly after the load instruction.

TABLE I
SLOWDOWN FACTOR WITH RESPECT TO NATIVE EXECUTION.

Benchmark	mips	crypt	sstep
hanoi-16	130	513	40423
hanoi-24	144	578	—
mmult-50	56	196	12288
mmult-100	108	379	—
mmult-150	128	449	—
mmult-200	138	485	—
mmult-250	143	503	—

IV. PERFORMANCE

In this section, we compare execution speed of native code, single-stepped native code, simulated code, and simulated code with data and code encryption. To test the speed of single-stepped native code, we have written a program that uses the `ptrace` system call to execute the tested application instruction at a time. We have evaluated the performance of a recursive Hanoi towers solver (with no output), and multiplication of square matrices of various sizes. The Hanoi towers solver has a relatively complex control-flow (doubly-recursive function), while matrix multiplication is data intensive.

The host programs have been compiled with gcc 4.3.2 in 64-bit mode with maximum optimizations (`-O3`), while the guest programs have been compiled with the mipsel-elf-gcc 4.2.4 cross-compiler, also at maximum optimization level. The tests have been run on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs, 64 GB of RAM running Linux kernel 2.6.27.3. We have used the `numactl` program to bind the program to a single CPU.

Table I shows the slowdown factor of single-stepped (sstep), simulated (mips) and simulated and encrypted execution (crypt) with respect to the native code running at full speed. We have run the Hanoi solver for $n \in \{16, 24\}$, and matrix multiplications for sizes 50, 100, 150, 200 and 250.

From the table, we can see that hardware-assisted single-stepping with `ptrace` has worst performance and causes the target program to run more than 10^4 times slower than the native version. Because of the extreme slow-down, we have been able to run only the smallest problem instances. This experiment also shows that single-stepping is not a viable method for on-the-fly decryption of code and data. The main cause of low performance are the high overheads of kernel involvement for every executed instruction.

Executing the applications with CSPIM incurs slow-down up to a factor of ~ 140. In the matrix multiplication test (mmult), the slowdown increases with the matrix size; the largest increase in slowdown happens at transition from $n = 50$ to $n = 100$. We believe that this is due to cache effects, as three 50×50 integer matrices use 30kB of RAM, which still fits into the CPU's L1 cache. For $n > 100$, matrices do not fit into the L1 cache anymore, so they compete also with the simulated CPU's instructions.[5] The table also shows that using encrypted memory accesses slows down the Hanoi benchmark

[5]Recall that instructions that the *guest* CPU executes are also *data* in the host's memory.

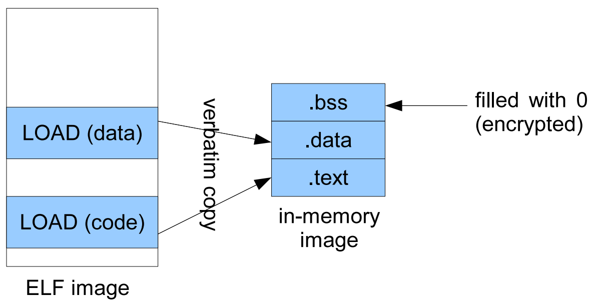

Fig. 4. Loading of ELF image where all loadable segments are encrypted.

by an additional factor of ~ 4, while the matrix multiplication benchmark is slowed down by an additional factor of ~ 3.5.

In summary, simulating a 32-bit MIPS CPU incurs ~ 140-fold performance penalty, which increases to ~ 580-fold performance penalty when encryption of code and data is enabled. In comparison, the secure context switch mechanism [14] incurs a 2.5-fold performance penalty on programs with 4 MB of protected memory space; the penalty would increase with larger sizes of the protected memory. Even though significantly faster than our method, secure context switch does not obfuscate the code, is not applicable to multi-processor systems, and is not resilient against observation within a virtual machine even on single-processor systems. Other program obfuscation methods, like VMProtect, are proprietary and have no published performance data.

V. Use cases

CSPIM executes MIPS I binaries on a virtual CPU. As such, it is already a powerful program obfuscation tool. Additionally, CSPIM supports encryption of instructions and data at the (simulated) physical memory layer, so only a single machine word (32 bits) is present in unencrypted form in the host's memory. Lastly, to provide an even larger degree of obfuscation, CSPIM can *simulate itself*, i.e., it is possible to construct *recursive simulations*. The provided code archive includes example programs demonstrating all these capabilities.

When encrypted execution is desired, the MIPS program must first be compiled into an ELF executable, which is in turn encrypted by the `elfcrypt` program, also provided in the code archive. The `elfcrypt` program encrypts all loadable ELF segments using the RC5 algorithm configured with 128-bit key and 32-bit block size. Such small block size guarantees that no loadable segment needs padding to the cipher's block size before being encrypted. For simplicity, `elfcrypt` uses ECB mode of operation, but there is no obstacle to using a more secure mode such as CTR, where the address of the simulated memory location would be used as the counter. Feedback and chained modes, e.g., CBC or OFB, are unsuitable because encrypted data cannot be randomly accessed in constant time.

CSPIM's ELF loader copies encrypted ELF segments verbatim into the simulated memory, and takes care that the uninitialized data section (`.bss`) is filled with encrypted zeros, as shown in figure 4. Once the encrypted ELF executable is embedded as data in the host's memory, its contents cannot be interpreted until a valid key is obtained. The key can, for example, be stored verbatim in the host's executable, generated by some algorithm, supplied by the user, or acquired over the network. Since the simulator incurs significant performance penalties, we envision that only the truly security-sensitive parts of the application would be run in the guest environment.

VI. Discussion

Our current scheme has several weaknesses that the attackers can use to their advantage when attempting to recover the encrypted code and data.

Plain-text ELF headers. The `elfcrypt` program encrypts only the contents of the loadable segments, while ELF headers and symbol tables remain in plain-text. In our proof-of-concept implementation, we have used ELF because it allows us to access objects in the simulated program by symbolic names instead of by absolute addresses. In production use, the host program would embed just the previously prepared in-memory image shown in figure 4, and access all objects by their absolute addresses. Since the presence of an ELF image(s) in the host's data segment would be easily detected, this approach additionally obfuscates the presence of the encrypted program.

Plain-text key. Once the key has been acquired, it *must* be present in plain-text in the host's memory. By finding and reading the key, the attacker will gain full access to the program's code and data. In the simplest case, the key will be as a sequence of bytes in the host's memory, so the attacker can extract all l-byte sequences, l being the key size, from the host and use them to attempt to decrypt the guest's code and data. This is an $O(n)$ process, n being the total size of the host's data. By *scattering* the key material through the host's memory, the complexity of the search process can be raised to $O(n^l)$ (exactly $\binom{n}{l}l!$ combinations, l being constant).

Plain-text simulator code. The simulator's code and functioning is fully exposed in the host code. Thus, an attacker could reverse-engineer the simulator code, find the memory layer functions responsible for encrypting and decrypting memory accesses, and use an OS-level or hypervisor-level monitor to observe the inputs and outputs of these functions. If the attacker in addition observes the accessed addresses instead of just data, he can know whether *every* instruction in the guest's code segment has been decrypted, and thus reconstruct the full guest's code. Such attack can be made significantly more complicated by using self-modifying code in the guest because successful decryption of the instruction at some address does not give any information about the number of different possible decryptions of the same address.

No tamper protection. Our scheme does not provide tamper-protection, i.e., it does not ensure that the guest code has not been manipulated before execution. Even if the attacker does not know the key, he can still maliciously manipulate the encrypted code or the encrypted data that the guest operates on. The former will lead to invalid instructions in the majority of cases, while the latter will allow the attacker to subtly manipulate the inputs and the outputs of the guest program. Tamper-protection cannot be implemented purely in software, and an orthogonal, hardware-based approach, such as Flicker [18], is needed.

Small block size. For reasons already explained in section V, the encryption block size is only 32 bits, which might facilitate dictionary and matching cipher attacks [19]. We do not consider this as a serious drawback because the weaknesses described above are much easier to exploit. Increasing the block size is possible, but it would make the compilation and decryption process slightly more complicated.

Orthogonality. CSPIM is orthogonal to existing obfuscation techniques [7], [8], [9], which can be used to additionally obfuscate the host *and* the guest code.

Nevertheless, none of these measures are perfect: they can only make the reverse-engineering process so tedious that the attacker will simply give up. With enough time and resources at hand, the attacker will *eventually* gain full knowledge about the guest application. Again, this weakness can only be countered by executing the simulator in a hardware-protected environment. Execution with hardware protection would also solve the problem of tamper protection: the results of the guest program would be signed with the private key that is otherwise inaccessible to other concurrently running programs and, indeed, the operating system (or hypervisor) itself.

VII. Conclusion and future work

In this paper, we have presented a technique that combines MIPS I CPU simulation and strong symmetric encryption (RC5 algorithm) to provide code and data protection for applications. Unlike existig approaches, which decrypt large portions of the application's code and data, our method never decrypts more than a *single memory word* (32 bits) at any time during the execution of the simulated CPU. This complicates the reverse-engineers' task because they have to penetrate several obfuscation layers to learn about the program's inner workings: the CPU simulator layer, the encryption layer, finding the encryption key, and deducing the encryption mode. The CPU simulator layer and the simulated program can additionally be obfuscated by existing program obfuscation methods.

Our simulator (CSPIM) is written in cross-platform C code and has been tested on Windows, Solaris and Linux operating systems, compiled with three different compilers for 32- and 64-bit mode. We have benchmarked its performance using a recursive towers of Hanoi solver and multiplication of square matrices, and established that the slow-down factors of simulation with encryption disabled and enabled are up to 150 and 580, respectively.

Although costly, our method provides both obfuscation and encryption, and, unlike secure context switch, is resilient to virtualization-based analysis tools. Furthermore, the decryption key does not need to be an integral part of the executable containing the protected code, which makes the protected code *completely* resilient to reverse-engineering until the key has been obtained.

Also, unlike secure context switch, our method is applicable to systems with multiple CPUs. In fact, the host program can use today's modern multi-core machines to its advantage: the guest program can be spawned to asynchronously run on another core, while the host program proceeds at full speed.

As part of our future work, we would like to investigate two directions: applying our method to protecting programs particularly susceptible to reverse-engineering (e.g., Java and .NET applications) and improving performance. Performance can, in turn, be improved in two ways: by improving the current simulator code, and by *parallelizing* it. Parallelization could, for example, emulate pipelined execution and spawn separate threads for on-the-fly encryption and simulation. Parallelization would complicate the reverse-engineers' task even more, because they would have to follow several concurrent control flows instead of one.

References

[1] "Amazon Elastic Compute Cloud," Available online., Accessed August 2009, http://aws.amazon.com/ec2/.

[2] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," in *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, 2005, pp. 2–13.

[3] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 51–62.

[4] VMProtect Software, "Vmprotect," http://www.vmprotect.ru/.

[5] Oreans technologies, "Themida," http://www.oreans.com/.

[6] Silicon Graphics Inc., "MIPS Assembly Language Programmer's Guide," 1992, part number 02-00036-005.

[7] P. Beaucamps and E. Filiol, "On the possibility of practically obfuscating programs towards a unified perspective of code protection," *Journal in Computer Virology*, vol. 3, no. 1, pp. 3–21, 2007.

[8] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.

[9] Y. Guillot and A. Gazet, "Semi-automatic binary protection tampering," *Journal in Computer Virology*, vol. 5, no. 2, pp. 119–149, 2009.

[10] J. Larus, "SPIM: A MIPS32 Simulator," Available online., Accessed August 2009, http://pages.cs.wisc.edu/~larus/spim.html.

[11] grugq and scut, "Burneye," Available online., Accessed August 2009, http://www.phrack.com/issues.html?issue=58&id=5&mode=txt.

[12] N. Mehta and S. Clowes, "Shiva," Available online, Accessed August 2009, http://cansecwest.com/core03/shiva.ppt.

[13] E. Filiol, "Strong cryptography armoured computer viruses forbidding code analysis: the BRADLEY virus," in *V. Broucek ed., Proceedings of the 14th EICAR Conference*, 2005.

[14] T. Morris and V. Nair, "Secure context switch for private computing on public platforms," in *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, 30 2008-Dec. 4 2008, pp. 1–5.

[15] "DarkParanoid virus description," Available online., Accessed August 2009, http://www.viruslist.com/en/viruses/encyclopedia?virusid=2622.

[16] "Engine of eternal encryption," Available online., Accessed August 2009, http://vx.netlux.org/vx.php?id=ee02.

[17] Željko Vrba, "Encrypted execution engine," *Phrack*, no. 63, Accessed August 2009, http://phrack.org/issues.html?issue=63&id=13&mode=txt.

[18] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for tcb minimization," in *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. New York, NY, USA: ACM, 2008, pp. 315–328.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.