Enhancements to a virtual machine based code encryptor

Nikos Mavrogiannopoulos
ESAT/COSIC
KU Leuven – IBBT

June 2011

Abstract
In this document we present the CSPIM virtual machine-based code encryptor, discuss open issues related to its Application Programming Interface (API), obfuscation issues, and propose enhancements. The discussed enhancements include a new, simpler, programming interface and a new obfuscation profile.

1 Introduction to CSPIM

CSPIM [7] is a MIPS I instruction set emulator, with the ability to interpret encrypted code [8]. This provides a level of obfuscation, that as we have shown in [4] requires an adversary with advanced capabilities to recover the plain code.

It operates by having a parent application execute an encrypted MIPS-compiled ELF binary. The run-time operation is very similar to a typical virtual machine operation and is shown in figure 1. The encrypted ELF binary is expanded initially to virtual machine memory and the virtual MIPS CPU is interpreting instructions by decrypting them on demand. At no point in time memory remains unencrypted.

The process of making such an application can be summarized to:

- Create the code to be protected separately from the main program. It must not contain library or operating system calls.
- Compile the code using a MIPS I cross compiler, but without linking to any standard libraries.
- Encrypt the generated binary using the CSPIM provided utility “elfcrypt”.
- Test the application with the CSPIM provided utility “runtorture”, and include it in the parent application. The latter process will be discussed later.

The process is depicted in figure 2.
However we have identified few issues in CSPIM that we believe they can be further improved. Our improvements span from the application programming interface (API), to the low level encryption algorithms and modes used.

2 Application programming interface

2.1 Current situation

The available API in CSPIM consists of several low level functions defined in CSPIM internal files such as `vm/cpu.h`, that enable the programmer to initialize the MIPS I virtual machine, and load an ELF file to be interpreted. A typical initialization of the virtual machine would be like:

```c
mips_init();
pcpu = mips_init_cpu(base_mem, MEM_SIZE, STACK_SIZE);
prepare_cpu(pcpu, ELF_FILE, HEX_KEY);
execute_loop(pcpu);
```

where `execute_loop` would be:
2.2 The issues and solutions

This however is a low level interface requiring application developers utilizing CSPIM to interpret an encrypted (or not) ELF binary, to handle parts of MIPS interpretation within their program. Moreover there is no easy way of communication between the main program and the interpreted code, and the interpreted application has no access to standard library’s functionality, hence reducing its potency. An example application that requires direct communication between the main program and the interpreted code is a license checking application as in figure 3. In that application the main program needs to transfer the license as entered by the user and the interpreted code needs to report the status back.

![Diagram](image)

Figure 3: An example license checking application utilizing CSPIM.

To solve those issues we created a higher-level library that would abstract those operations and decouple emulation details from the main application. To enable
communication between the two processes we used the MIPS syscall interface, and a callback function that will be fired once the interpreted application executes the \texttt{SYSCALL} instruction. That way based on the arguments of the system call, the parent application will act accordingly. The only internal functions exposed were for memory reading and writing to allow the parent application read and write to memory addresses within the interpreted application’s memory boundaries.

The interpreted application was provided a tiny library that allows calling a number of system and standard library calls, such as as calls for memory allocation and manipulation, etc, in order to ease porting of existing code snippets.

### 2.3 The new library API

The new API is shown below and is contained within the \texttt{libcspim.a} library archive. The header for this API is \texttt{cspim.h}.

```
typedef void* cspim_cpu_t;

/** Initializes a virtual CPU with given memory and stack size */
int cspim_cpu_init(cspim_cpu_t *pcpu, unsigned int memsz, unsigned int stacksz);

/** Prepare CPU for execution with optional encryption key. * The ELF file memory address provided must be valid until * deinitialization. */
int cspim_cpu_prepare_mem(cspim_cpu_t _pcpu, void* elf, unsigned int elfsz, const char *asckey);

/** The same as cspim_cpu_prepare_mem() but loads directly from a * file. */
int cspim_cpu_prepare_file(cspim_cpu_t _pcpu, const char *exename, const char *asckey);

/** Deinitializes virtual CPU. */
void cspim_cpu_deinit(cspim_cpu_t pcpu);

/** Execute until exception and report status to stdout. Handles only * SPIM syscalls. Returns 0 on successful termination. */
int cspim_execute_spim(cspim_cpu_t _pcpu);

/** Definition of the system call, callback function. */
typed int (*cspim_syscall_fn)(void* priv, unsigned int no, uint32_t arg1, uint32_t arg2, uint32_t arg3, uint32_t arg4, uint32_t arg5);

/** Executes until the number of loops (instructions) is reached. * If loops is given as -1, it executes until the program is * terminated. The callback function \texttt{fn} will be called on */
```
42 * system calls requested on the virtual CPU.
43 */
44 int cspim_execute(cspim_cpu_t _pcpu, int loops, void* priv, 
45 cspim_syscall_fn fn);
46 /**<
47 * Dumps all the registers in the virtual CPU
48 */
49 void cspim_mips_dump_cpu(cspim_cpu_t pcpu);
50 /**<
51 * Copy (potentially unaligned) data from host to the simulator (out), or from
52 * simulator to the host (in). In the case of failure, the state of the MIPS
53 * simulator has not been altered.
54 */
55 int cspim_mips_write(cspim_cpu_t pcpu, uint32_t dst, void *src, unsigned int n);
56 int cspim_mips_read(cspim_cpu_t pcpu, void *dst, uint32_t src, unsigned int n);

2.4 The new MIPS library API

The new MIPS library API is shown below and is contained within the libmips.a library archive. The header for this API is mips.h. It provides interface to the basic SPIM system calls, that were already included in CSPIM, and adds the ability to request arbitrary system calls, to be handled by the parent process. In addition to that new memory manipulation and allocation functionality was added.

#define USER_SYSCALL(x) (x+0x1f)

#define SYSCALL1( n, arg1)
#define SYSCALL2( n, arg1, arg2)
#define SYSCALL3( n, arg1, arg2, arg3)
#define SYSCALL4( n, arg1, arg2, arg3, arg4)
#define SYSCALL5( n, arg1, arg2, arg3, arg4, arg5)

/* Definitions of common functions */
void *memcpy(void *dest, const void *src, unsigned int n);
void *memset(void *s, int c, unsigned int n);
int memcmp(const void *s1, const void* s2, unsigned int n);
void * malloc(unsigned int size);
void free(void*);
void* calloc(unsigned int x, unsigned int s);

/* Definitions for SPSIM syscalls */

/* These constants must match the native system's constants! And these modes
* seem to be the only ones that are equal among unices (at least Solaris and
* linux). Therefore, open also adds O_CREAT if it detects writing mode. */
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR 2
#define SEEK_SET 0
#define SEEK_CUR 1
3 Encryption issues

CSPIM uses encryption to “scramble” the stored MIPS code. We used the term “scramble” because the purpose of the encryption is not to thwart a computationally bounded adversary from breaking the scheme but rather to divert his efforts to key recovery from the program code, which is considered a costly task [4].

The algorithm used is RC5 with a 32-bit block in ECB mode. However we believe that this combination of cipher and mode has issues, related to the role of encryption and other issues of practical nature. The decision of a 32-bit cipher block size is considered to be a weak design choice but is imposed by the constraints of existing tools. We elaborate on the issue in section 3.5.

3.1 Obfuscation issues

The combination of ECB mode and the 32-bit block size has issues because it cannot hide the code structure sufficiently. That is because in ECB mode identical instructions would encrypt to identical ciphertext. This is illustrated in figure 4 and figure 5.

3.2 Tamper-resistance issues

The ECB mode choice has other issues that relate to tamper-resistance of the code. Those are:

- ECB mode is not sufficient to prevent copy and paste attacks (e.g. copy program’s dummy operations such as NOPs over some critical check code).
- The combination of ECB mode with a 32-bit block size is susceptible to code book attacks.

Those limitations provide an adversary a tool to manipulate encrypted code without the knowledge of the key. Code for example could be overwritten by

---

1That matches the MIPS instruction size, which is also 32-bits.
Figure 4: Illustration of ECB mode of operation on an image (this example was taken from the Wikimedia Commons).

Figure 5: Illustration of ECB mode of operation on code.

using the ECB encryption of the NOP MIPS instruction, which can be found using statistical analysis of the encrypted code. This is a feasible task as code generated by GCC for MIPS CPU utilizes the NOP instruction often and in predictable positions.

3.3 Performance issues

The fact that CSPIM operates using a virtual machine adds the cost of instruction interpretation. Moreover the encryption and decryption operations for every memory access reduce the execution speed by a factor of 5x (see table 1). We thus can see that there is room for optimization and a faster cipher could improve the instruction loading time, which in turn would result to faster execution time.
3.4 Enhancements

3.4.1 Cipher selection

In order to improve the performance of the decryption process and also eliminate the dependency on a patented algorithm, such as RC5, we evaluated alternative ciphers that could be used as a drop-in replacement. This was not an easy task as very few published ciphers operate using a 32-bit block size. The requirement for a 32-bit block size initially looks arbitrary, but the advantage of such a cipher is that it can encrypt each individual MIPS I instruction (which is of 32-bit size), as a singleton. That way and because jumps are always occurring at instruction boundaries, the decryption of instructions is straightforward.

Cryptographic cipher designers, do not consider 32-bit block ciphers as secure, as they can easily be attacked using code book attacks, i.e., attacks where a codebook is generated with all possible inputs and outputs for a given key. This however is not a threat in our case. Assuming the model in [4], where the adversary is a reverse engineer that has access to a single encrypted executable with a given key. If that key is not reused for other purposes, the generation of a code-book would require the adversary to operate the decryption functionality of the code, which would give him the plain code anyway. Thus the generation of the code book would not provide him any additional advantage.

An ideal candidate for such a system would be a white-box cipher [9] but due to lack of any real-world ciphers matching this definition, we evaluate real-world ciphers. Any cipher that offers short-term protection against an adversary (based on the definition in [5]), would suite for this framework. That is because the encrypted code’s purpose is not to thwart a computationally bounded adversary from breaking the scheme but rather to diverge his efforts to a alternative methods of breaking the protection [4].

Based on [5], we could assume that 80-bit security level or more, fullfills our requirement. Hence our full requirements for the cipher are:

1. 32-bit block size;
2. Key size of more or equal to 80-bits;
3. Random access operation.

KATAN One of the ciphers that operate using 32-bit block size is KATAN [1]. Its key size is 80-bits and the cipher is optimized for embedded hardware. We implemented the cipher in software in the C language, but the performance comparing to RC5 was many orders of magnitude worse. Even though KATAN used simple operations per round, being an LFSR-based cipher, the high number of 254 rounds it requires, set the cipher at a significant performance disadvantage comparing to the 12 round RC5.
3.4.2 Cipher mode selection

To counter the tamper resistance and obfuscation issues identified in the previous sections we evaluated other cipher modes of operations. For that we set requirements based on the issues to be resolved and the available constraints.

1. Prevent deterministic modifications to the code;
2. Encryption of code should hide patterns, i.e., identical instructions should result to different encrypted blocks;
3. Random access of data must be guaranteed.

The Electronic codebook mode or ECB, does fulfill only the last requirement of random access. Cappaert in [2], is discussing the usage of counter mode (CTR), as seen in figure 6, for program encryption. This operates using an ever-increasing counter something that the virtual memory address of data naturally provide. It can provide random access of data on any memory location and has advantages over the ECB such as:

- Dependency of the data from the stored virtual memory address, to prevent copy-paste attacks;
- Patterns are hidden because ciphertext depends on data and counter.

However its error propagation is local, something that is not an issue when used in an authenticated cipher mode [3], although in our use-case it is a disadvantage. That is because we set as a requirement to prevent deterministic modifications to code. Even if the copy-paste property of ECB is no longer present because of the ciphertext dependence on memory addresses, the counter mode relies on XOR, a linear function, between the counter and the plain code. Hence the attacker can flip bits of the stored instructions in order to modify them in a deterministic way.

For this reason we utilized the RCTR mode, a modification to counter mode, displayed in figure 7. This operates using the counter principle as in counter-mode but without the linear dependency of ciphertext with plaintext, thus fulfilling
all the 3 requirements we set. The linear dependency of counter mode is being removed by mixing the counter early to the ciphertext, and applying encryption on the mixed output.

This mode keeps the advantages of the counter mode and eliminates the ability of deterministic modifications to the code, because the cipher decryption would provide no linear dependencies between ciphertext and decrypted code.

### 3.5 Implications to ELF file loading

Replacing however a simple encryption mode with an advanced one, provides benefits, but comes to a cost. This, in our case, relates to how encryption is performed on the ELF [6] file by “elfcrypt”, which we try to explain in the next paragraphs.

The ELF file format is organized in segments and sections, with each segment containing different sections, as in figure 8. A typical program loader reads the ELF information and loads to the program’s address space all segments marked with the PT_LOAD flag. This typically includes a segment that contains the application’s code which is by convention located in the .text section.

The ELF file segments at loading time, are copied to memory and aligned to memory page size. In typical files the stored segments are aligned with p_align value in the program headers, which is typically set to 4 bytes by the GNU linker. This alignment prevents the usage of a block cipher with size other than 4 because padding of the ending blocks and alignment of the initial block cannot be done without substantial changes to the linker.

The CSPIM “elfcrypt” utility purpose is to encrypt the application’s code. For that it parses the ELF file headers, and by using the information on the Program header, it encrypts all the loadable segments and replaces them with the encrypted data. This approach has the advantage that all the data intended to be in the process’ address space are encrypted, and thus they are easily decrypted by using a decryption hook on every memory access, after the data are
However because a loadable segment might contain the ELF headers and the program headers as well, loading of the encrypted program was impossible, because the ELF headers were encrypted and critical information such as the base address were not available. Decrypting them during loading time is possible when using the ECB mode, as the original CSPIM did, but the RCTR mode depends on the virtual memory addresses for encryption and decryption requiring the base address to be available prior to decryption.

The approach we followed to overcome that issue was to modify the behavior of “elfcrypt” to not encrypt the ELF headers, even if they belonged to a loadable segment. With this change, after the loader had parsed the headers, the virtual addresses were known and decryption of the rest of the segment using the RCTR mode would be possible.

4 Summary

We presented several optimizations to the existing emulation and obfuscation framework CSPIM. To test the effect of our optimizations we generated a real-world scenario of a complex function to be obfuscated and a parent program utilizing the obfuscated functionality. That involved two applications, one to produce MD5 checksums and one to feed data to the former. The MD5 checksum calculator was compiled using a GCC 4.2.4 MIPS cross compiler, and the feeder application was compiled using the native compiler.

The emulated code was executed as it was expected even though there was a significant performance degradation due to emulation layer. We display performance results in Table 1. At the first row we show the performance of the same program run natively, i.e., without the MIPS emulator being present, for comparison reasons.

We can see that encryption adds overhead to the emulator. However the encryption mode RCTR adds almost no overhead comparing to plain ECB mode.
Program Type | Execution time | Data processed | Execution throughput |
--- | --- | --- | --- |
Native CPU | 5 secs | 207 Mb | 41 Mb/sec |
MIPS virtual CPU | 5 secs | 442 Kb | 88.5 Kb/sec |
MIPS virtual CPU, RC5-ECB encrypted | 5 secs | 88 Kb | 17.5 Kb/sec |
MIPS virtual CPU, RC5-RCTR encrypted | 5 secs | 83.7 Kb | 16.8 Kb/sec |

Table 1: A performance summary of various tests using the CSPIM framework. The data processed indicates how many data were processed by the MD5 algorithm, which was compiled without any optimizations.

References


