

# A taxonomy of self-modifying code for obfuscation

(This is an updated version of the article. The published article is available at

<http://dx.doi.org/10.1016/j.cose.2011.08.007>)

Nikos Mavrogiannopoulos<sup>a</sup>, Nessim Kisserli<sup>a</sup>, Bart Preneel<sup>a</sup>

<sup>a</sup>*Katholieke Universiteit Leuven  
ESAT/SCD/COSIC – IBBT  
Kasteelpark Arenberg 10 Bus 2446  
B-3001 Leuven-Heverlee  
Belgium*

---

## Abstract

Self-modifying code is frequently used as an additional layer of complexity when obfuscating code. Although it does not provide a provable level of obfuscation, it is generally assumed to make attacks more expensive. This paper attempts to quantify the cost of attacking self-modified code by defining a taxonomy for it and systematically categorising an adversary's capabilities. A number of published methods and techniques for self-modifying code are then classified according to both the taxonomy and the model.

*Keywords:* self-modifying code, code obfuscation

---

## 1. Introduction

There are many reasons to protect a program's implementation details. Commercial entities generally do so in an attempt to preserve the secrecy of algorithms and protocols that are considered to provide a competitive advantage. While crackers and cyber criminals share this goal, they also seek to confound defense systems, postpone their detection and eventual tracing. The technique by which a program  $P$  is transformed into another, semantically equivalent yet harder to understand program  $P'$ , is known as obfuscation, the notion of which was first formalized by Barak et al. [6].

The term obfuscation is applied to a broad range of techniques, covering both syntactic and semantic transformations with results of varying efficacy. Obfuscation methods include the simple renaming of variables, stripping of comments, flattening of the control-flow graph, replacement of instructions with equivalent ones, including use and emulation of entire instruction sets, and dynamic code creation. The latter approach, when changing existing code, is referred to as self-modifying code, and is of particular relevance to obfuscation because of the inherent additional complexity it introduces. This has also been seen as a barrier to its adoption as conventional programming pattern.

Unfortunately for those entrusting secrets to obfuscation, Barak et al. in [6] established its impossibility under a formal definition. However, despite theoretical shortcomings, obfuscation and self-modifying code are used in practice to raise the barrier for adversaries by commer-

cial entities [40, 14]. Their use also figures prominently in grayer areas of security such as bypassing anti-virus engines [28, 44, 43] and intrusion detection systems [41].

Given that obfuscation is no panacea, those wishing to use it must weigh its deployment cost against the effort required of an adversary to defeat it. The former can be quantified, to some extent, in terms of initial cost of developing the obfuscation, subsequent program execution slowdown, increased file size, and complications to user support. The increased complexity of combining self-modifying code with obfuscation and its effect on an adversary cannot currently be adequately ascertained due to a lack of systematic categorization and precisely defined terminology with which to characterize it.

A taxonomy clearly defines relevant terms enabling the unambiguous, reproducible categorization of items it purports to describe. It is in this spirit that we present our taxonomy of self-modifying code for obfuscation and apply it to a number of published methods. We classify these based on the attributes of a toolset available to an adversary (defined in Section 3.1.1). We do not, however, attempt to capture more subjective and variable aspects such as the adversary's skill or familiarity with low-level details of the execution environment. Neither the associated cost of developing a tool, nor the technical expertise required to effectively use it are specified; these vary greatly over time as available reverse-engineering and debugging tools become both more effective and easier to use.

Because self-modifying code weakens the distinction between program code and data, we begin the paper with

an introduction to code and its use on modern computers, followed by an overview of self-modifying code. The core of the document, Section 3, defines code obfuscation, describes our classification methodology, and includes several real-world examples of obfuscation techniques using self-modifying code. Further discussion of the taxonomy and observations on its relation to the surveyed examples can be found in Section 4. We conclude in Section 5.

## 2. The nature of self-modifying code

### 2.1. An architectural overview

To convey the nature of self-modifying code we briefly examine modern computer architecture. Typical processors based on the von Neumann central processing unit (CPU) architecture execute instructions from main system memory. These instructions are different for each processor and their size varies from a few to several bytes [24, 3]. Although in a typical CPU architecture the main memory may be divided in sections or segments, this distinction is often only logical and the CPU will execute control transfers (i.e. jumps) to memory addresses in different sections. In the Intel x86 [24] family of CPUs<sup>1</sup> for example, applications are executed using the “flat” memory layout. Memory is reserved, into which the application’s code and data are copied, before the system transfers control to it. The memory the application “sees” is virtual, its layout imposed by the system, and initially, only partially mapped to the underlying physical memory. A typical memory layout, under an operating system such as GNU/Linux, is shown in Figure 1. The code section (often referred to as text) holds the application instructions while the BSS<sup>2</sup> and data sections hold uninitialized and initialized application variables respectively. Two special memory sections are the “heap” and “stack”. The former being a variable memory section from which the application’s requests for extra memory are fulfilled. The stack memory section serves multiple purposes including the storage of operating system environment variables, command line arguments, compiler-specific variables, and current program execution context values (such as the saved instruction pointer) used by the CPU’s function call instruction during execution.

Despite the separate depiction of code section from the remainder of the memory sections in Figure 1 such segregation is purely logical. Code placed in any other section will be executed by the processor<sup>3</sup>. This lack of distinction

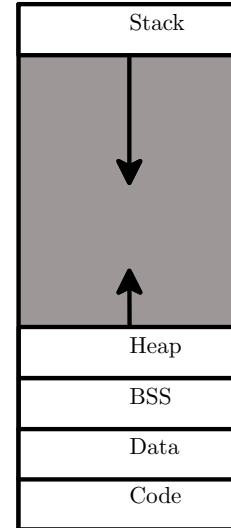


Figure 1: Process memory organization.

between code and data allows programs to store data in memory for later interpretation as instructions, or in other words, it allows self-modification of programs.

### 2.2. Global overview of self-modifying code

Our paper’s main focus is self-modifying code for obfuscation purposes. However, the following paragraphs briefly illustrate other real-world applications of self-modifying code to provide a broader overview of the technology.

*Self modifying code for optimization and extensibility.* A trend in software that peaked with Sun’s Java language, was the notion of just-in-time compilation [5], usually of abstract machine instructions. Unlike a compiler which translates code to CPU instructions, the just-in-time compiler is part of an interpreter and operates by generating code at run-time. That idea was used by Massalin in [31] to design a proof of concept operating system using self-modification for optimization. Even some mainstream programming languages allow applications to replace and execute their code via code-generation. These include Perl through its eval function, COBOL with the ALTER keyword and a C language extension called “backtick” C [35] allowing dynamic code generation.

*Debugging, anti-debugging and tamper resistance.* Several debuggers modify the target application’s code. The debugger by Vanegue et al. [49], for example, injects its code into the debugged process. Other designs such as gdb [42] also modify the target’s code to implement breakpoints, by replacing the target code with a software interrupt instruction (on x86 architectures). Anti-debugging techniques [13, 37] often take advantage of such self-modifying behavior to thwart debugging. Additionally, self-modifying code has been used to improve software tamper resistance techniques [20].

<sup>1</sup>Other architectures such as MIPS [45] or ARM [3] operate in a similar fashion, but details may vary.

<sup>2</sup>Its name (Block Started by Symbol) is a relic of history with little bearing on its use today.

<sup>3</sup> New generation CPUs such as the x86-64, ARMv6 [3, 25] support access permissions for memory pages allowing system and applications to make a distinction between executable memory and memory containing data.

*Code injection and control flow modification.* The lack of distinction between code and data allows for code injection attacks such as “Stack smashing”, “Format string attacks” and many others [2, 32]. In these attacks, the injected code is written to a program’s memory as input data to which control is subsequently transferred. The majority of attacks of this kind are prevented today at the hardware level by marking writable memory pages as non-executable [21, 3, 25]. Attack techniques evolved however, circumventing the protection offered by marked pages by modifying a program’s control flow in so-called arc-injection attacks such as return-to-libc and return oriented programming [9, 39, 8],

### 3. Self-modification for code obfuscation

Self modification has a long history of use in software obfuscation. Early DOS programs used it to hide their copy protection functionality, and shareware programs obfuscated their internal structures against modification [22]. It was even used to prevent software from executing on a competitor’s operating system [38] by storing the code in an obscure format and correctly extracting it at run-time.

In the next sections, we introduce a taxonomy for obfuscation techniques using self-modifying code. We describe the addressed threat model, and discuss our classification method before applying it to the published techniques we survey.

#### 3.1. Code obfuscation

##### 3.1.1. Threat model

The notion of obfuscation was formalized by Barak et al. [6]. Their formalization depends on an obfuscator satisfying the “virtual black box” property, which guarantees that an adversary with access to an obfuscated program should not learn anything more than is possible via oracle access to the original, unobfuscated program. In other words, access to the program code should not provide more information than can be gained by observing the program’s outputs. Under this definition, obfuscation is proved impossible [6]. Combined with work showing self-modified software to be semantically equivalent to non-self modified software [7], self-modifying techniques are not expected to provide a provable level of obfuscation.

However, self-modification for obfuscation is usually expected to raise the adversary’s cost of reverse-engineering by the cost required for deobfuscation. This is what we attempt to quantify in this study, based on a classification of attack tools, similarly to the approach in [4]. The threat in this case is an adversary with debugging, dis-

assembling and emulation<sup>4</sup> tools available and seeks to reverse-engineer a program. Additionally, the adversary is assumed to have all required information for executing the software in question<sup>5</sup>, but not necessarily any knowledge of the protection method used.

##### 3.1.2. Adversary’s capabilities

In a typical developer’s environment [10] the available debugging and disassembling tools are not perfect and have limitations that can be exploited by self-modifying code. The limitations from a reverse engineering perspective are discussed in [49]. Of particular relevance here is their reliance on a static analysis of the executable to obtain control flow information.

Not all debuggers have this limitation, thus we model an adversary described by a list of gradually increasing capabilities. This is shown in Table 1, and models adversaries with different capabilities ranging from a simple disassembler, to the availability of an ideal debugger as in [30] or specialized tools designed to attack a particular method such as a tool to circumvent self-checking code [48].

In the first category we model an adversary with static analysis tools, such as a disassembler. Category II models the typical debugging tools available to software developers, such as gdb [42]. Those tools are limited by their reliance on static analysis for information about the debugged program and are thus unable to handle self-modifying code<sup>6</sup>. Category III models the ideal debugger in [30], that has no practical issues when handling self-modifying code. The last category models an adversary able to create or obtain specialized tools with which to attack the binary. We note however that a protection method’s complexity may make it more efficient to create such tools than to mount a lower-category attack as illustrated in Section 4.2.

#### 3.2. Attributes of the taxonomy

This section explores the use of self-modifying code for obfuscation. We survey a number of published techniques, academic and otherwise, and classify them according to the common attributes described below. Taken together, we call the combination of an obfuscation technique’s attributes its *protection profile*.

The presentation of each attribute in the taxonomy is kept intentionally succinct, while a discussion of some of the

<sup>4</sup>We will not distinguish between a debugger and an emulator as there is no fundamental difference in their capabilities for the purposes of reverse engineering.

<sup>5</sup>For this reason protection methods based on secret information, such as a password, will be ignored.

<sup>6</sup>In our tests, even single stepping instructions overwritten by others was not detected by the debugger which appears to cache code, incorrectly displaying the modified pages.

Category	Capability	Description
I	Disassembler	Merely obtain an Assembly language description of the program, derived through static analysis.
II	Debugger with no ability to handle self-modifying code	A debugger limited by its reliance on static analysis to obtain control flow information.
III	Debugger that handles self-modifying code	A debugger that operates correctly when parts of code are modified.
IV	Specialized tools	The adversary is able to obtain tools specifically targeting the protection technique.

Table 1: Adversary’s tool capabilities

more nuanced aspects, as well as the relation of the attributes to the selected adversary’s capabilities in Table 1 is provided in section 4.

### 3.2.1. Ways of concealment

The body of a program can be obfuscated in different ways, including:

- In *slices*: The program is treated as a collection of parts, possibly separately or differently obfuscated. In practice, slices are generally program functions.
- As a *singleton*: The program is treated as a single block to be obfuscated as a whole.

For our purposes, we will consider any part of a program as a *program slice*.

### 3.2.2. Encoding

Code can be encoded in many different ways, ranging from *simple* substitutions to more *complex* transformations, such as encryption or compression. Methods vary according to the granularity of encoding, space and run-time overheads. Replacing individual instructions with random data for example, while highly granular, requires a reverse mapping of data back to the original instructions. Although a keyed block cipher only requires a key, its use of fixed-sized data blocks reduces its granularity.

### 3.2.3. Visibility and exposure

We use the term visibility to describe the amount of code obfuscated. It includes *complete visibility*, when no part of the program is obfuscated, *partial visibility*, when parts are obfuscated but others remain in clear, and *no visibility* when a program is fully obfuscated. Intuitively, a visible slice is one which may be read (and executed) whereas a slice with no visibility is unreadable until deobfuscated.

We further classify exposure over the lifetime of a program as complete, or temporary, depending on whether program

slices are re-encoded after use or not. A *completely exposed* program slice is one which is decoded for execution but not subsequently re-encoded. A *temporarily exposed* program slice is re-encoded after use, minimizing its exposure to an adversary. A program’s exposure type may partially depend on the encoding scheme used.

## 3.3. Methods

### 3.3.1. Kanzaki’s method

In order to increase the barrier to understanding software, Kanzaki et al. [27, 26] describe a method of obfuscating program instructions by overwriting them with *dummy* ones. The idea can be summarized as having parts of software which, at run-time, restore other component’s dummy code with both the original code, and code for restoring the dummy instructions once the original has executed. The proposed system works on Assembly code and consists of:

1. Determining the positions to be protected<sup>7</sup>, as well as those of the Hiding and Restoration routines.
2. Replacing original instructions with dummies.
3. Creating and inserting Hiding and Restoration routines.
4. Complication of the routines.

The first step is the most complex one due to the need to determine the positions of the Hiding and Restoration routines. To determine those positions the following rules are given:

- All paths leading to dummy instructions must include a restoration routine.
- No path between a restoration routine and a dummy instruction can include a restoration routine.
- All paths between a hiding routine and a dummy instruction must contain a restoration routine.

<sup>7</sup>This may be random or specified.

- All paths from a dummy instruction to the end of the program must include a hiding routine.

The replacement of instructions is done in a way that preserves the length of the instruction to be replaced. This avoids any need for relocation<sup>8</sup> of the program once the instructions are replaced. After the instructions are hidden they are marked using an Assembly label, and two routines are generated. One will restore the instruction under label to the original, and the other will hide it again.

As a final step, an obfuscation of the transformation routines is performed. Those are modified to refer indirectly to the memory address to be modified, and in addition some modification might be applied. The modifications proposed are to store a different value of the label and add a sequence of instructions (such as addition, subtraction, etc.) that in the end will result to the correct memory address.

**Summary:**

Implementation:

- No public implementation is available.

Can be used to hide:

- The program in slices.

Encoding:

- Replaces instructions with random data.

Exposure:

- Code is partially visible and can only be temporarily exposed.

### 3.3.2. Madou’s method

Madou et al. propose a rewriting engine [29] that will rewrite functions based on a template before execution. A pseudo-random number generator (PRNG) is used to encode the rewriting templates, and techniques such as opaque variables<sup>9</sup> from [17] are relied on to hide the PRNG seed. In effect the PRNG is used as a stream cipher with the seed as key. The rewriting engine is embedded in the program.

In detail, this method replaces functions of a program with a template containing the function with some instructions replaced by randomly generated data. Uses of the function are updated to reference code calling the rewriting engine with details of the function’s entry point and an “edit script”. This is data in memory containing information required for reconstructing the original code using the

<sup>8</sup>Programs have virtual addresses, and jumps within programs refer to those. If program memory is modified by the insertion of some bytes, all references to addresses must be updated.

<sup>9</sup>Opaque variables in this context are variables that have some property that is known to the obfuscator but the deobfuscator cannot deduce. See [17] for more information. It is not known whether variables with this property exist.

template. The rewriting engine uses the edit script and the function’s location to modify the template, recreating the original code.

The authors distinguish between One-Pass mutations and Cluster-Based mutations.

*One-Pass Mutations.* With this technique each function in the program has its own template and edit script. The function’s initial code calls the rewriting engine with information about the template and function address. Before executing, the initial function is rewritten and the rewriting code is itself overwritten by the original code.

*Cluster-Based Mutations.* In this approach functions are grouped according to their code similarity and a common template is generated for functions in the same class. As before, each unique function is replaced with calls to the rewriting engine using the corresponding edit script and entry point. However since the template code is shared the original function is generated in memory and the call to the rewriting engine updated to point to it.

In order to avoid analysis of the edit scripts, they are encrypted using a seeded PRNG. The key is generated at run-time as an opaque variable to prevent its easy recovery.

**Summary:**

Implementation:

- No public implementation is available.

Can be used to hide:

- The program in slices.

Encoding:

- Replaces instructions with random data. The decoding instructions (template) are protected using a stream cipher.

Exposure:

- Code is partially visible and after decoding is completely exposed.

### 3.3.3. Shell-code hiding

A defense against unintentional code injection is program input validation, ensuring accepted data conforms to certain rules or formats, such as alpha-numericness etc.

To counter that in [56] the authors explore ways to hide code for ARM CPUs [3] in instructions that are firmly alpha-numeric, in order to masquerade code in a text-looking format that passes any validation check for alphanumeric characters. The authors face the problem of having a limited number – only 13 – of ARM instructions that are within the printable limits of ASCII characters. This limited set of instructions lacks arithmetic operations and any branch instruction, hence causing problems in

writing any non-trivial piece of code, such as a worm or shell-code<sup>10</sup>. However by using self-modifying code the instruction set is increased with the set of bytes that can be written as an exclusive or (XOR) of a printable character with another one, since instruction XOR is within the printable character set. An illustration of the idea on how to generate null bytes required for the following instruction<sup>11</sup> using ARM assembly, is shown below:

```
mov r0, #0 ; byte representation: 0xe3 0xa0 0x00 0x00
```

To construct this instruction, the authors used the following code, adapted from [56], simplified and added comments for clarity:

```

1          ; Load in register 1 the 16 bits that are
2          ; in label .Linstr + 2 bytes.
3          ; r1=0x0180
4  ldrh   r1, .Linstr+2
5
6          ; XOR register 1 with the value 0x0180
7          ; and place output in register 1.
8          ; r1=0x0000
9  eor   r1, r1, #384
10
11         ; store in label .Linstr the value of register 1
12  strh   r1, .Linstr
13 .Linstr:
14  .byte 0xe3, 0xa0
15  .byte 0x80, 0x01 ; new value = 0x00, 0x00

```

The latter code generates the new instruction in place of the old instruction in line 9 and executes it. All the instructions used in the code snippet above, such as LDRH, EOR and STRH are within the printable character set. This technique is used by the authors to generate code that is entirely textual.

**Summary:**  
Implementation:  

- No public implementation is available.

Can be used to hide:  

- The entire program as a singleton.

Encoding:  

- Replaces instructions with ones that have a corresponding code in the printable character set, or ones that can be recovered using instructions from the printable set.

Exposure:  

- Code is partially visible and after decoding becomes completely exposed.

<sup>10</sup>Shell-code is code that is intended to be injected to running applications on a machine, with the goal of executing a shell.

<sup>11</sup>For simplicity in this example it is assumed that only null bytes cannot be used.

### 3.3.4. Methods used in Burneye

Written by team-teso [46], Burneye<sup>12</sup> is a tool to obfuscate ELF binaries [47], optionally password protecting them and imprinting them such that they only run on specifically fingerprinted hosts. We will restrict our description on the aspects of the tool that hide code and involve self-modifying code. The tool provides alternative options for protecting code, that can be combined.

- Simple scrambling: A Galois linear feedback shift register is used to encipher the original code.
- Encryption with a secret: The contents of the program are encrypted using the RC4 [36] cipher with a user-supplied password as keying material. The actual key derivation mechanism uses salting to discourage rainbow table attacks.
- Fingerprinting: Various characteristics of the system on which the binary is to be executed are collected and the code is encrypted with this information as key.

The first option, encrypts the executable with a simple cipher and no secret, and the transformation is reversed at execution time. The second and third option use a different cipher, with a secret part to encrypt the binary. The secret part is required for the execution phase, and in second option is given by a password prompt, while at third option the running system’s characteristics are used for generating a key. Although there are cases where the last two options will increase the security level, in our model they provide no extra security since we assume knowledge of the secret information by the adversary.

A similar to the first option in operation method is described in [50, Section 12.17], by Viega et al. That method uses the RC4 cipher to encrypt an executable program and decrypts it at run-time.

**Summary:**  
Implementation:  

- A public implementation is available in [46].

Can be used to hide:  

- The entire program as a singleton.

Encoding:  

- Uses a linear feedback shift register (LFSR) based or RC4 stream cipher to encrypt code.

Exposure:  

- Code is non-visible and after decoding becomes completely exposed.

<sup>12</sup>The rootkit used during the infamous compromise of three Debian servers in 2003 was obfuscated using Burneye.

### 3.3.5. UPX compression

In [34], Oberhummer and Lazlo describe UPX, a multi-platform executable file compressor. UPX stands for the Ultimate Packer for eXecutables and is mainly used for reducing the size of executables in restricted systems, but is also used by viruses and other executables to obfuscate their code. UPX can use a variety of compression algorithms, such as UCL [33], LZMA<sup>13</sup> or the proprietary NRV [33] algorithms. The default and fastest decompressor being NRV.

Prior to compression, UPX processes code (called filtering) to increase the compression level. That processing is architecture specific. On x86, for example, relative addresses in JMP and CALL instructions are replaced by absolute, causing a larger byte match for the compressor. Following this, the UPX packer tool compresses the target executable using the specified compression algorithm, and prepends the unpacker to the executable.

At run-time the unpacker takes control, decompresses the original program to memory, reverses the filtering process, and transfers control to it.

#### Summary:

##### Implementation:

- A public implementation is available in [34].

##### Can be used to hide:

- The entire program as a singleton.

##### Encoding:

- Uses a compression algorithm, i.e., LZMA, UCL or NRV, to obfuscate code.

##### Exposure:

- Code is non-visible and after decoding becomes completely exposed.

### 3.3.6. Methods used in Shiva

Shiva is an GNU/Linux ELF executable [47] encryptor written by Shawn Clowes and Neil Mehta first presented at CanSecWest in 2003 [15] to forward the state of the art in binary obfuscation on UNIX platforms. While the original presentation gives a high-level overview of Shiva, details of its workings were first publicly presented by Chris Eagle at BlackHat 2003 [19].

Shiva encrypts a program in blocks using the Tiny Encryption Algorithm (TEA) [55] and decrypts them on-demand to ensure the whole binary is never fully exposed in memory for dumping to disk. In addition, it allows the TEA encrypted blocks to be further encrypted using AES [18] and decrypted with a runtime user-supplied password. Several other anti-debugging techniques are used.

The code of the original program is being replaced by encoded data consisting of three blocks:

- Block 1, the Shiva code.
- Block 2, data used by Shiva.
- Block 3, The protected code in TEA encrypted blocks (possibly wrapped in a further AES encryption if password protected).

The outer encoding is done using simple instructions such as XOR and ADD operations. At program start-up the binary is decoded producing the three blocks in memory. After this step the Shiva code, takes control and starts decryption on demand.

Shiva encrypts not only the instructions used by the program's code, but also encodes all metadata available through the ELF file information. It uses anti-debugging techniques such as jumping into the middle of instructions, polymorphic code generation, and also implements its own runtime environment with memory maps and on-demand paging. Unused memory is filled with the INT 3 software interrupt<sup>14</sup> and when called, will trigger code that will do decryption and mapping of the missing page for execution to resume.

The memory mapping is implemented using encrypted blocks containing meta-information about the original program's memory layout as normally given by the ELF's program header table. These meta-blocks are encrypted with different keys, each of which is reconstructed dynamically, used, then cleared. The key-reconstruction functions are themselves differently obfuscated for each binary.

Finally, Shiva replaces some instructions in the original program with the INT 3 software interrupt storing the original operands and emulating the original instruction via the ptrace interface during the interrupt. This robs reverse engineers of the certainty of having captured all instructions in a code block.

#### Summary:

##### Implementation:

- No public implementation is available.

##### Can be used to hide:

- The entire program as a singleton.

##### Encoding:

- Relies on both simple encodings (XOR and ADD) and cryptographically strong methods, such as block ciphers TEA and AES.

##### Exposure:

- Code is non-visible and is only temporarily exposed.

<sup>13</sup>LempelZivMarkov chain algorithm.

<sup>14</sup>The INT 3 interrupt in a x86 GNU/Linux system causes the executed program to receive a SIGTRAP signal. That signal is handled by appropriate program code.

### 3.3.7. Methods used in Cryptexec

Cryptexec is an x86 code obfuscator, published by Vrba [51] in phrack 63. It encrypts the a number of program functions using the CAST cipher [1] in ECB<sup>15</sup> mode with a given key. Similarly to Shiva in Section 3.3.6 it tries to prevent expanding the executable code in memory. For that during decoding it operates like a virtual machine decrypting and executing the encrypted code instruction by instruction. It maintains its own private stack space from which it allocates two execution contexts, one for the encrypted code, the other for the decoder. Decoding is performed as follows:

- The decoder fetches the next instruction from the encrypted code.
- The data is decrypted.
- The plaintext is disassembled, yielding the original instruction.
- The instruction is executed using the encrypted code context.

Cryptexec provides an API to access encrypted functions thus making it easy to provide access to the encrypted of parts of the executable. In addition it ensures that during decoding no more than two single instructions are decrypted in memory at any time.

**Summary:**

Implementation:

- A public implementation is available in [52].

Can be used to hide:

- The program in slices.
- The entire program as a singleton.

Encoding:

- Relies on the cryptographic block cipher CAST in ECB mode.

Exposure:

- Code is non-visible and is only temporarily exposed.

### 3.3.8. CSPIM

CSPIM is a modification of the Cryptexec method published by Vrba in [54]. Instead of obfuscating the original program instructions, the code to be obfuscated is compiled to the MIPS I instruction set [45] and encrypted using the RC5 cipher in ECB mode using a 32-bit block size. The MIPS I instruction set is selected due to its simplicity, thus requiring a simpler emulator, and the small block size leads to a fine-tuned decryption granularity. During

<sup>15</sup>Electronic Codebook Mode. The simplest mode to use a block cipher.

run-time a virtual machine decrypts and emulates the encrypted code instruction by instruction. The operation is very similar to Cryptexec, but with an added emulation layer, that allows for portable obfuscated code that will be interpreted by a platform depended virtual machine.

**Summary:**

Implementation:

- A public implementation is available in [53].

Can be used to hide:

- The program in slices.
- The entire program as a singleton.

Encoding:

- Relies on the cryptographic block cipher RC5 with a 32-bit block size in ECB mode.

Exposure:

- Code is non-visible and is only temporarily exposed.

### 3.3.9. Aucsmith's tamper resistant software

Aucsmith discusses in [4] the issue of software tamper resistance and proposes an implementation to achieve it. We give a quick overview of the method and how self-modifying software is used within this framework.

The threat model in this application is explained using three categories:

1. The malicious threat originates outside the system. The perpetrator is using the network to “get in” the system.
2. The perpetrator has access of software running on the system. The attacker is bounded by the operating system. In this category the perpetrator is someone who had at one time access to the system.
3. The perpetrator has complete access to the system. He is the “owner” of the system. This is further split into:
  - (a) No specialized analysis tools, such as debuggers and disassemblers are available.
  - (b) Software tools such as debuggers and disassemblers are involved.
  - (c) Hardware analysis tools, such as processor emulators are available.

The categories this framework targets against are 2 and 3. The 3<sup>rd</sup> category is claimed to be handled up to attacks involving hardware analysis tools. The basic principles this implementation is based on are

- To disperse secrets in time and space.
- Obfuscation of interleaved operations. This is having the full functionality split and executed by successive iterations or rounds of the code. To prevent

discovery of the interleaved components the usage of self-modifying code is suggested.

- Have an installation unique code such as having each instance of software unique elements.
- Interlocking trust. The correct performance of components should depend on the correct performance of others.

The mixture of those principles is suggested to guarantee tamper resistance.

The architecture is composed of “Integrity Verification Kernels” (IVKs), that follow the above principles and an “Interlocking Trust Mechanism”. The former are “protected” code segments, that ensure the integrity of the process. The Interlocking Trust Mechanism is a mechanism to allow IVKs verifying other IVKs. Without getting into the details of the mechanisms we will describe their usage of self-modifying software.

The IVKs are described to be encrypted and self decrypt their body during execution. The cipher used should ensure that as decryption of a code section occurs, other code sections should be encrypted. It is also suggested to re-use memory locations for different operations.

This is the first study that advises the usage of self-modifying code to protect software. There are no formal results such as a metric on how security can be evaluated based on the threat level they protect against, but nevertheless it is an important source of ideas for software protection.

**Summary:**

Implementation:

- No public implementation is available.

Can be used to hide:

- The entire program as a singleton.

Encoding:

- Relies on cryptographically strong methods.

Exposure:

- Code is non-visible and is only temporarily exposed.

### 3.3.10. Cappaert’s method

Cappaert et al. in [12, 11] proposes ways to avoid static analysis of executable code, focused on code encryption. The authors distinguish between bulk decryption, where a loader does the decryption of all the executable code at run-time, and on-demand decryption mode. The main focus of the proposal is the on-demand decryption mode, where functions are decrypted at run-time, and re-encrypted them after usage.

In this scheme “crypto guards” are introduced. Those are routines that will decrypt a function’s code based on the integrity of other function’s code. A cryptographic hash is used to obtain the key from another function’s code. This cryptographic hash will provide from arbitrary length data a fixed size byte array that can be used as a key with a symmetric cipher. Each function is protected with a key that is derived from the code of the caller, and re-encrypted when the function returns.

The authors of this scheme faced the problem of having multiple callers of a function. In that case they suggest on deriving a key from all the callers. However in this scheme the real caller of the encrypted function will be in clear whilst the other callers will be probably encrypted. This has the disadvantage of having to decrypt all the possible callers, in order to derive the decryption key<sup>16</sup>.

In addition to limiting the available code to a debugger, this approach offers protection against modification of the binary code. Any change in a function will cause an incorrect decryption of some other function.

**Summary:**

Implementation:

- No public implementation is available.

Can be used to hide:

- The program in slices.

Encoding:

- Relies on cryptographically strong methods.

Exposure:

- Code is non-visible and is only temporarily exposed.

## 4. Discussion

In the following sections we elaborate on the use of self-modifying code for obfuscation in the context of the presented taxonomy and the structure of the methods surveyed, summarized in Table 2. We discuss the concealment, encoding methods, visibility and exposure, and their effect on the defense against different adversaries.

### 4.1. Ways of concealment

The method of concealment, whether a singleton or in slices, has no direct effect on the overall quality of obfuscation. A weak relation exists however, between the method and the degree of exposure. Methods that do not

<sup>16</sup>The “Surreptitious Software” book [16] described a modified version of the Cappaert et al. algorithm called the `OBFCCKSP`. This tries to solve the multiple callers problem by moving the decryption of a function  $F$  to the callers of the caller. The code of all the direct callers of  $F$  is being used as decryption key of  $F$ .

use a virtual machine to emulate the obfuscated code, such as Kanzaki’s, Aucsmith’s and Cappaert’s [27, 12, 4], operate on code slices in order to offer temporary exposure. However virtual machine based methods such as Cryptexec, CSPIM and Shiva [51, 54, 15], demonstrate that singleton encoding methods can be combined with temporary exposure, by interpreting arbitrary (Turing-complete) programs, instruction by instruction.

Hence we see that even if the ways of concealment have no impact on the obfuscation, some techniques might impose a particular concealment method due to operational requirements.

#### 4.2. Encoding

The encoding method perceives the notion of code obfuscation, as a complex transformation of code to be stored in a binary file. To consider a method “secure”, devising a decoding method should be more costly than mounting an attack of a higher adversary category. For example, a decoder using byte to constant value XOR, after the method is realized may be cheaper to attack creating a specialized tool which recovers the clear code for disassembly than mounting an attack with a category III debugger (see Table 1 on page 4). Encoding methods, if used in isolation, by definition protect against the adversary of category I, i.e., the static disassembler.

##### 4.2.1. Simple methods

The simple encoding methods, such as instruction replacement and XOR or addition modulo integers, are invertible transformations used to obfuscate code. They are relatively understandable and used to conceal small amounts of code, generally supplementing other protection methods.

##### 4.2.2. Complex methods

Complex encodings such as encryption or code compression can protect the code completely from static analysis tools, such as disassemblers of category I. Almost all techniques using complex methods employ efficient algorithms such as the RC4 [36] stream cipher, block ciphers in ECB<sup>17</sup> mode, or compression methods with a fast decompressor. Methods using cryptography favor speed of decryption over cipher strength as testified by the prevalent use of “weak” modes of operation such as ECB. The cryptographic algorithms used need only be sufficiently strong

to ensure breaking the cipher is harder than reverse engineering the cipher and discovering the key. The algorithm’s speed is accorded far more importance. The inclusion of instructions for high-performance encryption in newer CPUs [23] might render methods previously deemed prohibitively expensive [11] feasible.

#### 4.3. Visibility

Visibility is a measure of method’s protection against static analysis. We distinguished two types of visibility, partial and none. In the former, parts of the code are obfuscated leaving others in the clear, while the latter type ensures all code is obfuscated.

Methods offering partial visibility leave a percentage of code in the clear. Consequently, relevant information may yet be extracted, even by a category I adversary with a disassembler. A high-level instruction such as the `printf` libc function call may contain a dozen or more assembly instructions and obfuscating a few may not sufficiently hide the code’s functionality. Replacing 10% of the instructions at random with dummies gives an adversary a 90% probability of finding a valid one at each check. By increasing the percentage of replaced instructions however, schemes like Kanzaki’s and Madou’s require duplication of large amounts of code.

Thus methods offering partial visibility can only defend probabilistically against an adversary seeking to reverse engineer the obfuscated code. Methods offering no visibility obfuscate the entire program code. While the effectiveness of obfuscation transformations varies, as abstract methods they offer full protection against static code analysis.

Visibility whether partial or none applies to category I adversary, by definition.

#### 4.4. Exposure

Exposure is a measure of a method’s protection during program execution. Although all surveyed techniques obfuscate a program’s code, some deobfuscate it in its entirety for execution, exposing it completely, while others do so only temporarily. We discuss differences between the two approaches and the impact on their levels of protection, assuming the usage of an encoding method.

---

<sup>17</sup>ECB exposes some structure of the plaintext as identical plaintext blocks yield the same encrypted blocks. In practice this has not been problematic as the decoder remains an easier target of attack [19].

Table 2: A summary of the taxonomy of self-modifying code for code obfuscation. The methods marked with ♣ have a public implementation available. The check mark(✓) or commenting text is used when a technique suits the corresponding attribute.

Method	Concealment		Encoding		Visibility		Exposure	
	In slices	Singleton	Simple	Complex	None	Partial	Temporary	Complete
<i>Kanzaaki</i>	✓		Code re-placed by random data			Depends on the percentage of re-placed instructions	Code decoded and re-encoded after execution	
<i>Madou</i>	✓			Code re-placed by random data, de-coding using PRNG		Depends on the percentage of re-placed instructions		✓
<i>Shell-code hiding</i>		✓	✓			Code that generates new code		✓
♣ <i>Burreye</i>		✓		LFSR or RC4 encryption algorithms	✓			✓
♣ <i>UPX</i>		✓		LZMA, UCL or NRV compression algorithms	✓			✓
<i>Shiva</i>		✓		TEA or AES encryption algorithms	✓		Virtual Machine	

Table 2: (continued)

Method	Concealment		Encoding		Visibility		Exposure	
	In slices	Singleton	Simple	Complex	None	Partial	Temporary	Complete
♣ Cryptexec	✓	✓		CAST encryption algorithm	✓		Virtual Machine	
♣ CSPJM	✓	✓		RC5 encryption algorithm	✓		Virtual Machine	
Aucsmith	✓			Any encryption algorithm	✓		Code decoded and re-encoded after execution	
Cappaert	✓			Any encryption algorithm	✓		Code decoded and re-encoded after execution	

#### 4.4.1. Complete exposure

Several methods only protect code against static attacks, before execution. If concealed as a singleton, program code is decoded in its entirety and stored in memory, available to an adversary. If, on the other hand, it is concealed in slices, only those decoded for execution are exposed. As the program continues to run, more slices are exposed until it is available in its entirety, as code is executed<sup>18</sup>.

Such methods prevent disassemblers from displaying any useful information and hamper debuggers with no support for self-modifying code. In the days of anti-virus software statically scanning binaries for known-signatures, viruses relied on such tricks to escape detection [43].

Completely exposed programs can be made available in clear to an adversary, even of category II, using a memory dump<sup>19</sup>. If the memory dump is taken after decoding, the full program code will be available to the adversary. For the reasons above, we note that the protection level of methods that completely expose code, is against adversaries with tools of category I.

#### 4.4.2. Temporary exposure

Methods that only allow temporary exposure such as Cryptexec, Kanzaki, Cappaert’s etc., take a piecemeal approach to encoding programs. Either instructions or whole functions are recovered at runtime, on-demand, and re-encoded immediately after use, ensuring that at no single moment of execution is the entire code available in the clear. In addition to preventing disassemblers from displaying any useful information, this method performs constant self-modification, for as long as protected functions are called. Moreover, it prevents adversaries at any point during execution from obtaining an overview of the whole program. Only the executing function or part and required data are available in clear.

For this reason we claim that the protection of this method is against adversaries with tools of category II or lower. Category III adversaries are also discouraged by depriving them from a direct global overview of the code.

#### 4.5. Relation to the adversaries

Table 3 summarizes the effects of our taxonomy’s attributes on the different adversaries defined in Section 3.1.1. The included remarks correspond to the discussions in the previous sections. The *concealment* criterion is omitted as it has no direct bearing on the overall security level, as discussed in Section 4.1.

<sup>18</sup>Non-executed code is obviously not decoded.

<sup>19</sup>Operating systems typically provide ways to take a memory dump of a process for debugging purposes. The GNU/Linux operating system, for example, saves a program’s registers and memory in a core file –suitable for examination with a debugger– upon receiving a SIGSEGV signal.

## 5. Conclusion

We have presented a taxonomy of self-modifying code for obfuscation and a model characterizing different levels of attackers in terms of the types of tools required to mount each attack. We used the model and taxonomy towards a primary goal of establishing a terminology and framework within which more precise discussions of self-modification for obfuscation may be pursued. Further secondary results include better motivating an obfuscator’s choice of components and their use in a protection scheme, as well as highlighting under-developed areas of analysis for the simultaneous exploitation by obfuscators, and implementation by attackers. Taken together, the combination of attributes listed in the taxonomy, namely concealment, encoding, visibility, and exposure, constitute an obfuscation technique’s protection profile. We show, perhaps unsurprisingly, that a protection profile combining complex encoding, no visibility, and temporary code exposure provides the highest protection against analysis from the defined adversaries.

As table 2 indicates, many of the obfuscation techniques employ robust protection profiles, requiring specialized tools to attack them. However, the fact that most have been successfully attacked highlights the ultimate futility of obfuscation. In a cat and mouse game, programs are obfuscated involving complex methods such as self-modifying code, to hide pieces of code for a period of time before they become irrelevant, and attacks with varying attack tools are performed to uncover the obfuscated code. Ultimately specialized tools are implemented to uncover obfuscated programs, but the custom nature of the tools may mean a change in the obfuscation’s protection profile necessitates a rewrite of the attacking tool. This can be seen at the attack on Shiva [19] which was subsequently modified by its authors to foil the custom deobfuscator.

We see our findings do not contradict the impossibility result of Barak et al. [6], nor the semantic equivalence of self modified software with non-self modified software [7]. We base our analysis on a modelling of the imperfection of today’s real-world tools available for reverse-engineering. Hence a subjective factor in the modelling of “difficulty” for reverse-engineering, is introduced, in a way that this study depends on the current-day technology. This dependence however provides the ability to assess methods that are used in practice, despite the negative theoretical results.

For that we have seen that obfuscation raises the bar of protection against our modelled adversaries of categories I and II. An adversary of category I, that is an adversary with static-analysis tools, that inherently has no ability to follow code transformations, and cannot reverse engineer programs encoded using a complex encoding method without devising a specialized tool that reverses the transformation. Likewise an adversary of category II, i.e., dynamic-analysis tools with restrictions that due to

Adversary	Description	Encoding		Visibility		Exposure	
		Simple	Complex	None	Partial	Temporary	Complete
I	Disassembler	Reverse engineering is easy	✓	✓	Some code is available	✓	✓
II	Debugger with no ability to handle self-modifying code					✓	
III	Debugger that handles self-modifying code					Prevents global overview	
IV	Specialized tools						

Table 3: The taxonomy attributes and their effects on protection level against adversaries. The adversaries' categories correspond to Table 1 on page 4 and the check mark(✓) indicates protection against this adversary.

bugs or limitations fail to follow code transformations, also require specialized tools to evade methods that offer temporal exposure. Those specialized tools by our definition would come at the expense of requiring an adversary of a higher category, being category III or IV.

Thus we have made apparent that current-day protection techniques are depending on the scarce availability of tools that resemble an “idealized” debugger and the difficulty of devising a custom decoding mechanism. The wider availability of category III tools that pose no issues in following self-modifying code would certainly render many of the discussed techniques irrelevant and possibly drive the technology to protection methods that exploit different tool limitations and restrictions.

## 6. Acknowledgments

The authors would like to thank Andreas Pashalidis, and the anonymous referees for their comments which improved the manuscript. This work was supported in part by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) SBO project, the Research Council K.U.Leuven: GOA TENSE (GOA/11/007), and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

## References

- [1] Adams, C., May 1997. The CAST-128 Encryption Algorithm. RFC 2144 (Informational). URL <http://www.ietf.org/rfc/rfc2144.txt>
- [2] Aleph-One, November 1996. Smashing the stack for fun and profit. Phrack (49).
- [3] ARM, 2009. ARM1136JF-S and ARM1136J-S: Technical Reference Manual. ARM.
- [4] Aucsmith, D., 1996. Tamper resistant software: An implementation. In: Proceedings of the First International Workshop on Information Hiding. Springer-Verlag, London, UK, pp. 317–333.
- [5] Aycock, J., June 2003. A brief history of just in time. ACM Computing Surveys 35 (2).
- [6] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K., 2001. On the (im)possibility of obfuscating programs. In: Lecture Notes in Computer Science. Springer-Verlag, pp. 1–18.
- [7] Bonfante, G., Marion, J., Reynaud, D., 11 2009. A computability perspective on self-modifying programs. In: 7th IEEE International Conference on Software Engineering and Formal Methods - SEFM 2009. Hanoi Viet Nam.
- [8] Buchanan, E., Roemer, R., Shacham, H., Savage, S., Oct. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In: Syverson, P., Jha, S. (Eds.), Proceedings of CCS 2008. ACM Press, pp. 27–38.
- [9] c0ntex, 2005. Bypassing non-executable-stack during exploitation using return-to-libc. Tech. rep.
- [10] Canonical, 2010. UBUNTU 10.04.
- [11] Cappaert, J., Kisserli, N., Schellekens, D., Preneel, B., 2006. Self-encrypting code to protect against analysis and tampering. In: 1st Benelux Workshop on Information and System Security (WISec).
- [12] Cappaert, J., Preneel, B., Anckaert, B., Madou, M., Bosschere, K. D., 2008. Towards tamper resistant code encryption: practice and experience. In: ISPEC'08: Proceedings of the 4th international conference on Information security practice and experience. Springer-Verlag, Berlin, Heidelberg, pp. 86–100.
- [13] Cesare, S., 1999. Linux anti-debugging techniques (fooling the debugger).
- [14] Cloakware, 2011. Cloakware. <http://www.irdeto.com/cloakware/>.
- [15] Clowes, S., Mehta, N., 2003. Shiva advances in ELF binary encryption. URL <http://cansecwest.com/core03/shiva.ppt>
- [16] Collberg, C., Nagra, J., 2009. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional.
- [17] Collberg, C., Thomborson, C., Low, D., 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Principles of Programming Languages 1998, POPL98. pp. 184–196.
- [18] Daemen, J., Rijmen, V., 2002. The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- [19] Eagle, C., 2003. Strike/counter-strike: Reverse engineering shiva.

- [20] Giffin, J. T., Christodorescu, M., Kruger, L., 2005. Strengthening software self-checksumming via self-modifying code. In: The 21st Annual Computer Security Applications Conference. IEEE Computer Society, pp. 23–32.
- [21] Grevstad, E., 2004. CPU-based security: The NX bit. Earthweb: Hardware.
- [22] Gruq, Scut, 2007. Armouring the elf: Binary encryption on the unix platform.
- [23] Gueron, S., January 2010. Intel Advanced Encryption Standard (AES) Instructions Set. Intel.
- [24] Intel, March 2010. Intel 64 and IA-32 Architectures Software Developers Manual. Intel.
- [25] Intel, 2010. Intel 64 and IA-32 Architectures Software Developers Manual. Intel.
- [26] Kanzaki, Y., Monden, A., Nakamura, M., 2004. A software protection method based on instruction camouflage. In: IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (Japanese Edition), J87-A(6):755767. pp. 47–59.
- [27] Kanzaki, Y., Monden, A., Nakamura, M., Matsumoto, K., 2003. Exploiting self-modification mechanism for program protection. In: in Proc. 27th IEEE Computer Software and Applications Conference. pp. 170–179.
- [28] Lemos, R., 2008. Researchers race to zero in record time. <http://www.securityfocus.com/news/11531>.
- [29] Madou, M., Anckaert, B., Moseley, P., Debray, S., Sutter, B. D., Bosschere, K. D., 2005. Software protection through dynamic code mutation. In: Proceedings of the 6th International Workshop on Information Security Applications. Springer, pp. 371–385.
- [30] Madou, M., Anckaert, B., Sutter, B. D., Bosschere, K. D., 2005. Hybrid static-dynamic attacks against software protection mechanisms. In: Proceedings of the 5th ACM workshop on Digital rights management. DRM '05. ACM, New York, NY, USA, pp. 75–82.
- [31] Massalin, H., 1992. Synthesis: An efficient implementation of fundamental operating system services. Ph.D. thesis, Columbia University.
- [32] Newsham, T., September 2000. Format string attacks. Tech. rep., Guardent, Inc.  
URL <http://seclists.org/bugtraq/2000/Sep/214>
- [33] Oberhammer, M. F., 2004. Ucl data compression library.  
URL <http://www.oberhumer.com/opensource/ucl/>
- [34] Oberhammer, M. F., Molnar, L., 2010. Upx: The ultimate packer for executables.  
URL <http://upx.sourceforge.net/>
- [35] Poletto, M., Hsieh, C., Hsieh, W. C., Engler, D. R., Kaashoek, M. F., 1999. ‘c and tcc: A language and compiler for dynamic code generation. ACM Transactions on Programming Languages and Systems 21, 21–2.
- [36] Rivest, R. L., 1992. RSA Data Security: the RC4 encryption algorithm.
- [37] Schallner, M., 2006. Beginners guide to basic linux anti anti debugging techniques. Code Breakers Magazine 1.
- [38] Schulman, A., September 1993. Examining the windows AARD detection code. Dr. Dobb’s.
- [39] Shacham, H., Oct. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: De Capitani di Vimercati, S., Syverson, P. (Eds.), Proceedings of CCS 2007. ACM Press, pp. 552–61.
- [40] Skype, 2011. Skype. <http://www.skype.com>.
- [41] Song, Y., Locasto, M. E., Stavrou, A., Keromytis, A. D., Stolfo, S. J., 2007. On the infeasibility of modeling polymorphic shellcode. In: Proceedings of the 14th ACM conference on Computer and communications security. CCS ’07. ACM, New York, NY, USA, pp. 541–551.  
URL <http://doi.acm.org/10.1145/1315245.1315312>
- [42] Stallman, R., Pesch, R., Shebs, S., 2010. Debugging with GDB: The gnu source-level debugger. Ninth Edition, for gdb version 7.2.50.20101209.
- [43] Szor, P., 2005. The Art of computer Virus Research and Defence. Symantec Press.
- [44] Szor, P., Ferrie, P., 2003. Hunting for metamorphic. Tech. rep., Symantec Security Response.
- [45] Technologies, M., 2001. MIPS32 Architecture for Programmers.
- [46] TESO, T., 2002. Burneye elf encryption program 1.0.1.  
URL <http://www.packetstormsecurity.org/groups/teso/indexdate.html>
- [47] Tool Interface Standards (TIS), 1993. Executable and Linkable Format (ELF).
- [48] van Oorschot, P., Somayaji, A., Wurster, G., 2005. Hardware-assisted circumvention of self-hashing software tamper resistance. IEEE Trans. Dependable Secur. Comput. 2 (2), 82–92.
- [49] Vanegue, J., Garnier, T., Auto, J., Roy, S., Lesniak, R., 2007. Next generation debuggers for reverse engineering.
- [50] Viega, J., Messier, M., Spafford, G., 2003. Secure Programming Cookbook for C and C++. O’Reilly & Associates, Inc., Sebastopol, CA, USA.
- [51] Vrba, Z., August 2005. cryptexec: Next-generation runtime binary encryption using on-demand function extraction. Phrack (63).
- [52] Vrba, Z., 2005. cryptexec: source code.  
URL <http://zvrba.net/software/cryptexec.tar.gz>
- [53] Vrba, Z., 2010. CSPIM: source code.  
URL <http://zvrba.net/software/cspim-1.0.tar.gz>
- [54] Vrba, Z., Halvorsen, P., Griwodz, C., 2010. Program obfuscation by strong cryptography. Availability, Reliability and Security, International Conference on 0, 242–247.
- [55] Wheeler, D. J., Needham, R. M., 1994. TEA, a Tiny Encryption Algorithm. Springer-Verlag, pp. 97–110.
- [56] Younan, Y., Philippaerts, P., November 2009. Alphanumeric RISC ARM Shellcode. Phrack (66).