Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Covert Computation — Hiding code in code through compile-time obfuscation☆

Sebastian Schrittwieser [a,*], *Stefan Katzenbeisser* [b], *Peter Kieseberg* [c], *Markus Huber* [c], *Manuel Leithner* [c], *Martin Mulazzani* [c], *Edgar Weippl* [c]

[a] *Vienna University of Technology, Favoritenstraße 9–11/188, 1040 Vienna, Austria*
[b] *Security Engineering Group, Darmstadt University of Technology, Hochschulstraße 10, 64289 Darmstadt, Germany*
[c] *SBA Research, Favoritenstraße 16, 1040 Vienna, Austria*

ARTICLE INFO

Article history:
Received 29 July 2013
Received in revised form
25 September 2013
Accepted 30 December 2013

Keywords:
Code obfuscation
Side effect
Code steganography
Semantic-aware malware detection
Compile-time obfuscation

ABSTRACT

Recently, the concept of semantic-aware malware detection has been proposed in the literature. Instead of relying on a syntactic analysis (i.e., comparison of a program to pre-generated signatures of malware samples), semantic-aware malware detection tries to model the effects a malware sample has on the machine. Thus, it does not depend on a specific syntactic implementation. For this purpose a model of the underlying machine is used. While it is possible to construct more and more precise models of hardware architectures, we show that there are ways to implement hidden functionality based on side effects in the microprocessor that are difficult to cover with a model. In this paper we give a comprehensive analysis of side effects in the x86 architecture and describe an implementation concept based on the idea of compile-time obfuscation, where obfuscating transformations are applied to the code at compile time. Finally, we provide an evaluation based on a prototype implementation to show the practicability of our approach and estimate complexity and space overhead using actual malware samples.

1. Introduction

During the last decade, malware detection has become a multi-billion dollar business and an important area in academic research alike. Static analysis, still the predominant technique for client based malware detection (commonly known as virus scanners), has not changed much during the last years. Current virus scanners almost entirely rely on signature based detection mechanisms (Christodorescu and Jha, 2004; Griffin et al., 2009). Malware, on the other side, has evolved significantly throughout the years and often uses sophisticated code obfuscation techniques in order to make detection more difficult. Encryption, polymorphism as well as metamorphism are commonly deployed to defeat signature based detection mechanisms by hiding the malicious functionality in data sections of the binary that look different for each instance of the malware.

To increase detection rates of obfuscated malware, new paradigms of malware analysis have been proposed. Semantic-aware malware detection, which was first introduced by Christodorescu et al. (2005), aims at solving some of

☆ This paper is an extended version of the conference paper (Schrittwieser et al., 2013).

* *Corresponding author*. Tel.: +43 1 5053688.
E-mail addresses: sebastian.schrittwieser@tuwien.ac.at (S. Schrittwieser), skatzenbeisser@acm.org (S. Katzenbeisser), pkieseberg@sba-research.org (P. Kieseberg), mhuber@sba-research.org (M. Huber), mleithner@sba-research.org (M. Leithner), mmulazzani@sba-research.org (M. Mulazzani), eweippl@sba-research.org (E. Weippl).

0167-4048/$ – see front matter © 2014 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cose.2013.12.006

the limitations of signature-based detection strategies by using so-called templates which define malicious behavior independently of its actual implementation. This approach makes the malware detection system more resistant against some types of obfuscating transformations such as *garbage insertion* (Collberg et al., 1997) and *equivalent instruction replacement* (De Sutter et al., 2009). However, a major limitation of this approach is its dependency on an accurate model of the underlying hardware (i.e., the microprocessor). In order to be able to evaluate the maliciousness of a sequence of processor instructions this model has to be detailed enough to map all effects on the hardware's state.

In this paper we show that exactly this fundamental prerequisite for semantic-aware malware detection is difficult to achieve. We introduce a concept called Covert Computation that is based on the idea of implementing program functionality in side effects of the microprocessor that are not covered by a simple machine model. In contrast to packer-based obfuscation which hides code in data sections that cannot be evaluated in static analysis scenarios, we go one important step further in this paper by hiding (malicious) code in real code. The main advantage of this approach over previous ones is that hidden functionality is not identifiable for syntactic malware detectors and extremely difficult to detect with semantic analysis techniques. In detail, the main contributions of this paper are:

- We introduce a novel approach for code obfuscation called Covert Computation, based upon side effects in today's microprocessor architectures. It hides (potentially malicious) code in legitimate code.
- We provide a comprehensive collection of side effects for Intel's x86 architecture and show how they can be used to hide (potentially) malicious functionality in executables.
- We describe a proof-of-concept implementation of our obfuscation technique that is performed at compile-time.
- We finally evaluate the security of our obfuscation approach against semantic-aware malware detection, measure the performance based on our prototype and provide a theoretical discussion on the effects of this obfuscation technique on real-life malware samples.

The remainder of the paper is organized as follows: In Section 2 we discuss related work in the area of malware obfuscation as well as malware detection. In Section 3 we introduce side effects of Intel's x86 instruction set and describe how they can be used to hide malicious functionality inside harmless looking code. In Section 4 we propose our concept of compile-time obfuscation and present a prototype implementation based on the LLVM compiler infrastructure. An extensive evaluation of our concept is described in Section 6. Finally, we summarize the main contributions of our paper and draw conclusions in Section 7.

2. Related work

Today's malware obfuscation approaches often follow the simple concept of hiding malicious code by packing or encrypting it as data that cannot be interpreted by the machine (Nachenberg, 1997). At runtime, an unpacking routine is used to transform the data block back into machine-interpretable code. Polymorphism (Song et al., 2007) and metamorphism (O'Kane et al., 2011) can be seen as improvements to the packer concept aiming at making automated malware detection more difficult. Another variant of packing was introduced by Wu et al. (2010). Their approach – called *mimimorphism* – encodes the program's code as harmless looking code, which is not detectable with previous concepts (such as entropy analysis) as the packed code appears to be code itself. Resulting binaries follow an unobtrusive statistical distribution of instructions and thus are able to trick malware detectors that work on the syntactical layer. However, this concept would not withstand a semantic code analysis. Even though the packed code looks like real code, it is just a sequence of functionally unrelated instructions without any semantic meaning.

On the other side of the arms race the detection and analysis of packed malware has been studied for many years. Many approaches are based on static code analysis. Encrypted code is identifiable based on entropy analysis as shown by Lyda and Hamrock (2007). Bruschi et al. (2006) described an approach for detecting self-mutating malware by matching the inter-procedural control flow graph of software against malware samples. The authors argue that despite its self-mutating nature, the control flow graph of this type of malware is still characteristic enough for reliable detection.

In contrast to the detection of packer-based obfuscation, the analysis of the actual semantics of code was proposed in literature using different approaches. The idea of using model checking for detecting malicious code was proposed by Kinder et al. (2005). Christodorescu et al. (2005) described the concept of semantic-aware malware detection, aiming at matching code with predefined templates specifying malicious behavior; matching of malicious code still works even if the actual implementation of the malicious behavior slightly differs from the reference implementation in the template. Dalla Preda et al. further formalized the approach of semantic-aware malware detection in 2007 (Dalla Preda et al., 2007) and 2008 (Dalla Preda et al., 2008). An important aspect of this type of malware detection is that is heavily depends on the quality of the model of the underlying hardware as the effects of a sequence of instructions has to be matched against the effects of a malware template. The first theoretical discussion on the idea of forcing a detection system into incompleteness was presented by Giacobazzi (2008). However, no practical approach of this idea was given in the paper. Moser et al. (2007) discussed the question whether static analysis alone allows reliable malware detection. The authors argue that semantic-aware detection systems are only effective against malware that is not protected against this particular analysis method and prove their claim with a new binary obfuscation schema that successfully prevents malware identification even by semantic-aware detectors. The paper concludes that simple obfuscation techniques can reliably hide the purpose of a program's code, and thus clearly shows the limits of static analysis.

Another approach against the threat of malware is to dynamically analyze the behavior of software in order to identify malicious routines (Willems et al., 2007). Sharif et al.

(2008) use Windows API call monitoring for deobfuscation. Malware detection using symbolic execution was put forth by Crandall et al. (2005). Bayer et al. (2009) describe TTAnalyze, a tool that records a program's system calls and Windows API function hooks in order to identify malicious behavior at runtime. While this software is mainly used by malware analysts to get an understanding of the malware's functionality, the concept was also extended to a preventive malware detection system that uses patterns of malicious behavior (Bayer et al., 2009), which were extracted by running malware in a controlled environment (Kolbitsch et al., 2009). Furthermore, dynamic malware analysis (Egele et al., 2012) has become an important concept to assist with processing malware samples on a large scale. While dynamic analysis is helpful for clustering mutated malware binaries according to their behavior, e.g., their system calls (Lanzi et al., 2010), this method still requires reverse engineering of obfuscated binaries in order to uncover crucial routines such as domain generation algorithms (Weaver, 2010), obfuscated encryption methods, or cryptographic keys (Chow et al., 2003).

3. Side effects

In this section we demonstrate that it is possible to implement sensitive parts of a program's functionality as side effects of an innocent looking piece of software. Our approach fundamentally differs from simple instruction replacements, which can be detected with semantic-aware malware detection systems. In contrast to simply replacing instructions with equivalent ones (e.g., MOV EAX, 0 with XOR EAX, EAX) our concept is much more subtle by moving the actual functionality as well as the data storage for intermediate results into side effects of instructions that are per se not equivalent to the original ones. In this paper, a comprehensive analysis of side effects in the x86 platform is conducted (see Table 1 for a complete list). For each side effect we explain (a) how it can be used to hide code, (b) how input data can be stored, and (c) where output data is put.

Table 1 – Side effects of the x86 architecture.

Hidden functionality	Host instruction	Side effect
Conditional jump	LOOP	CX/ECX
Short jump	LOOP	CX/ECX
MOV	LOOP	CX/ECX
	MOVS	ESI
	MOVS, EMMS	MMX/XMM
ADD	LODS	ESI
	REP MOVS	ESI
SUB	LOOP	CX/ECX
	LODS	ESI
	REP MOVS	ESI
INC	LODS	ESI
DEC	LODS	ESI
	REP MOVS	ESI
AND	RCL/RCR	Flags register
OR	RCL/RCR	Flags register
XOR	RCL/RCR	Flags register

3.1. Flags

In x86 the flags register is a 16 bit wide register that stores the processor's status (there exist successors with 32 as well as 64 bit width). Each bit (flag) of the register represents one status information. For instance, the carry flag is set to 1 if an arithmetic carry is generated by an arithmetic or bitwise instruction. Usually, these bits are used to store status information that is, in the following, evaluated in conditional control flow jumps. However, in the concept of Covert Computation we use the flags register to store input data for calculations that are entirely performed with the help of conditional control flow jumps.

Fig. 1 explains the concept based on a bitwise logical XOR operation. The basic idea is to map the four possible combinations of input values (00/01/10/11) by implementing two conditional jumps over the carry flag, each of it evaluating one input value. First, one input value is stored in the carry flag. This can be achieved by executing an instruction that sets the carry flag according to the input value. This instruction is followed by a conditional jump (JC). Then, the second input value is stored in the carry flag, again followed by a conditional jump. With this concept, all 4 possible output permutations can be mapped. To perform a conjunction over more than one bit, this process can be repeated. An example of an XOR operation over 32-bit input values without using any XOR instruction is given in Listing 1. The top part of the listing shows the original code and below the arrow a code fragment which implements the same functionality inside side effects of innocent looking code is given. The code uses the RCL (rotate through carry) instruction within a loop that runs 32 times. In each iteration one bit of the first input value (stored in EAX) is moved to the carry flag, followed by a conditional jump (JC) which splits the control flow so that depending on the input bit a different control flow path is taken. Then one bit of the second input value (stored in EBX) is moved to the carry flag (using the RCL instruction) and again a conditional jump is used to split the control flow. After performing the two conditional jumps, the program counter points to one out of four possible locations, which represent the four possible results of the XOR operation. Depending on which location is reached, the result is written to the carry flag with the help of either the STC (sets the carry flag to 1) or CLC instruction (sets the carry flag to 0). In the following iteration, this result bit is moved to EAX when the RCL instruction is executed for the next input bit. After 32 iterations, the final result can be found in EAX. In general, logical operations (AND, OR, and XOR) can be performed over the flags register without using the respective instructions.

The side effect in this concept lies in the rotation instructions RCL and RCR which rotate the register's content though the carry flag, thus it can be used to store input data. RCL shifts all bits towards more-significant bit positions. Further, the content of the carry flag is moved to the LSB, while the MSB of the register is moved to the carry flag. RCL performs the rotation towards less-significant bit positions. Thus, the carry flag is used to store the input values for a logical or arithmetical operation. The result of the operation is stored indirectly as the program counter's position within the

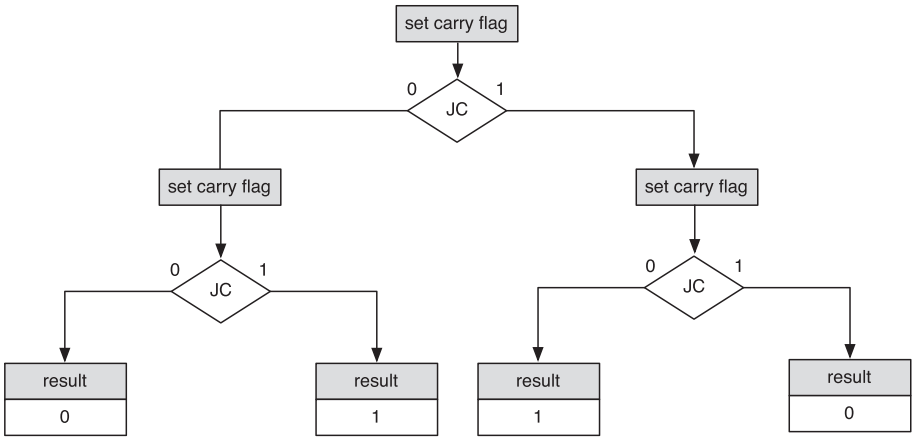

Fig. 1 – XOR using the carry flag.

control flow graph after executing the two conditional jumps. From there it can be copied to the flags register using dedicated instructions (STC and CLC), from where it is again moved to an output register (in our example: EAX) using the rotation instructions.

3.2. *LOOP Instruction*

In x86 the LOOP instruction uses a counter register (CX/ECX) that is decremented by one in each iteration. The loop terminates if this counter register contains the value 0. Otherwise, a jump to a location, which is specified by the operand of the LOOP instruction as a relative offset, is taken. The value stored in the counter register can be used in several ways to implement hidden functionality using the LOOP instruction. Listing 2 shows a conditional jump implemented using a LOOP instruction. Instead of making the jump decision depending on the zero flag (e.g., by using the TEST instruction), a side effect of the ECX in its role as the loop counter is exploited to archive the functionality of a conditional jump. Similarly, an unconditional short jump can be implemented, by moving a value unequal 1 to ECX.

Listing 3 shows a code fragment which implements the functionality of the SUB instruction inside side effects of the LOOP instruction. While the loop is executed, the values of EAX and ECX are swapped by the XCHG instruction. The ECX register serves as the counter register for the LOOP instruction and is decremented by one in each iteration. However, due to the XCHG instruction, the value stored in ECX constantly switches between the loop counter and the value of EAX. Thus the value of EAX is actually decremented in every other iteration of the loop. When the counter register ECX finally reaches 0, the value of EAX was decremented by the original value of ECX. The side effect is exploited as follows: Intermediate results are stored in the loop counter ECX, which carries out two tasks. The obvious functionality is that ECX decrements by one each time the loop's body gets executed. Combining the instruction with the XCHG instruction, however, leads to the effect that also the second operand gets decremented, thus the functionality of the SUB instruction can be imitated. The final result is also stored in the second operand. Note that in Listings 1 to 3 the value of ECX is modified. If the register is used at this location, its value has to be saved and restored.

3.3. *String instructions*

The MOVS, SCAS, CMPS, STOS, and LODS instructions are intended to operate on continuous blocks of memory instead of single bytes, words or dwords. In most cases, these

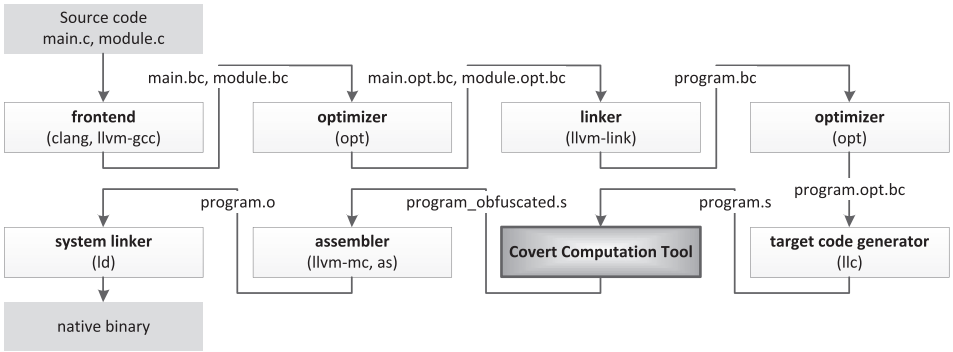

Fig. 2 – Intercepting the compilation process of LLVM.

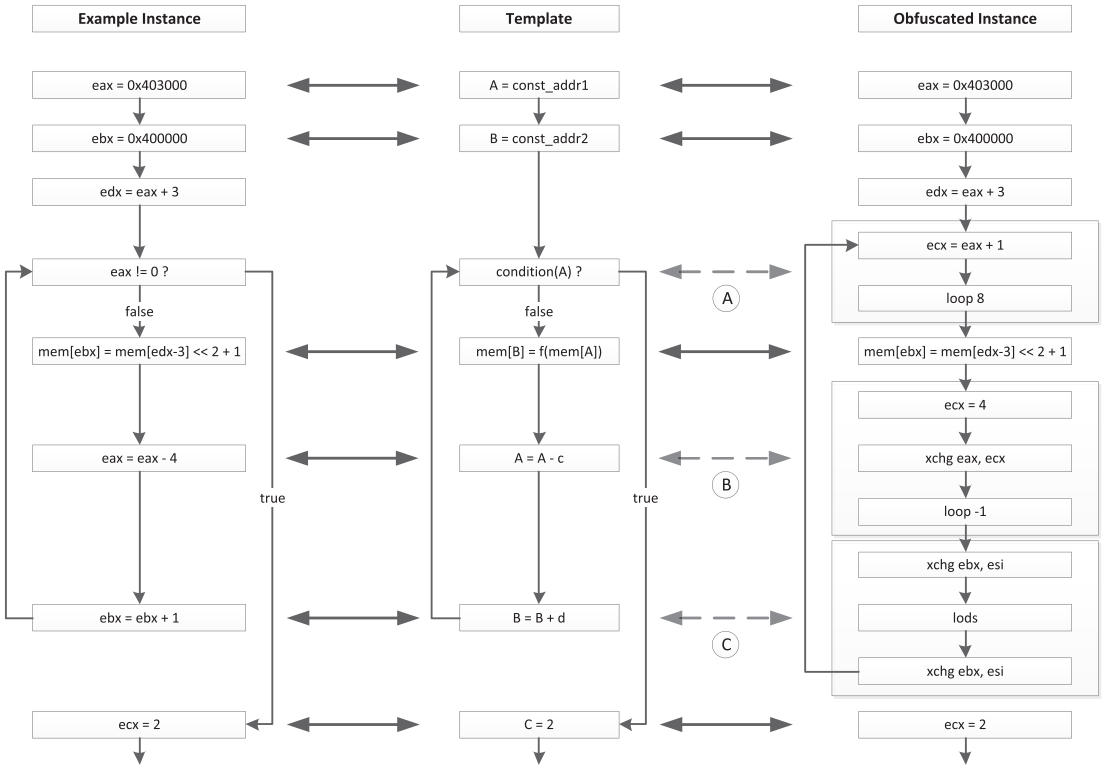

Fig. 3 – Resilience against semantic-aware malware detection.

instructions utilize implicit operands instead of programmer-defined registers. The implicit operands – if applicable for the specific instruction – are as follows:

- ESI as a pointer to the source block of memory
- EDI as a pointer to the destination
- ECX as counter (e.g., to specify how many elements to copy)
- AL/AX/EAX as a value for comparisons

Furthermore, the direction flag (which can be set with STD and cleared with CLD) determines whether ESI/EDI will be incremented or decremented after an operation. Each of the instructions modifies ESI, EDI or both. The REP prefix (as well as its siblings, REPZ/REPE and REPNZ/REPNE) is of further interest. This prefix, which is only applicable to string instructions, behaves in much the same way as LOOP: It repeats the given instruction ECX times. Without the REP prefix, string operations only operate on a single byte, word or dword. Before considering possible ways to repurpose their side effects, a short explanation of each of these instructions is given:

- MOVS moves (copies) bytes, words or dwords from the address pointed to by ESI to the address pointed to by EDI.
- SCAS scans the address pointed to by EDI for the value of AL/AX/EAX and sets the flags accordingly (this instruction is usually combined with REPE or REPNE to search for the first match or nonmatch of a given value).
- CMPS compares the value pointed to by EDI to the value at ESI and sets the flags accordingly.
- STOS stores the value at AL/AX/EAX into the location specified by EDI.

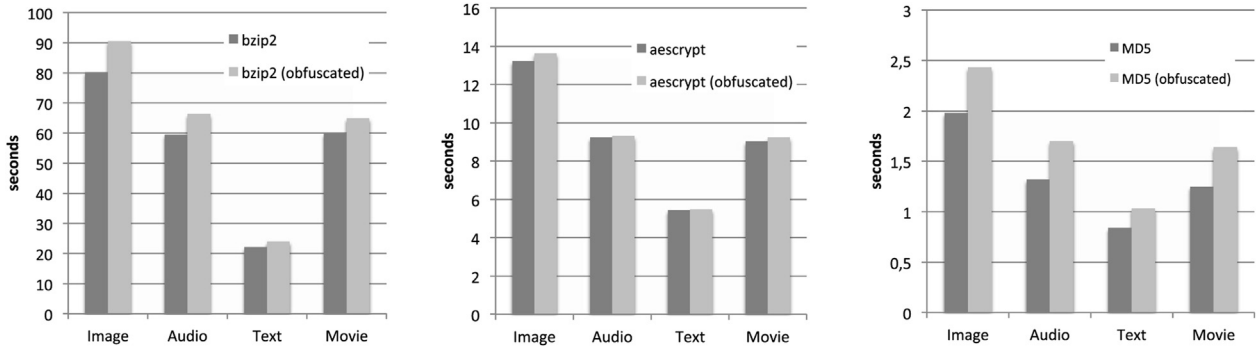

Fig. 4 – Performance analysis with aescrypt, MD5, and bzip2.

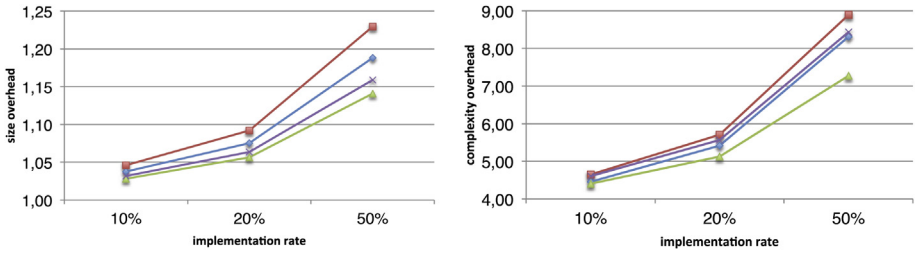

Fig. 5 – Theoretical evaluation of program (top) and complexity (bottom) overhead for four malware samples.

- LODS stores the value pointed to by ESI into AL/AX/EAX. This is the only string instruction where using REP is uncommon (if used at all).

The most obvious side effect is the modification of ESI, EDI, and ECX. By utilizing these properties, it is trivial to emulate ADD, SUB, INC, and DEC. Listing 4 shows a code fragment which implements the functionality of the INC instruction using side effects of the LODS instruction.

Note that this fragment is only applicable if the value of EAX points to a memory location that is accessible to the program. Most values will work; notably, 0 will not. This value will lead to a segmentation fault as it is not a valid memory address. Furthermore, the code overwrites the value in the ESI register, which has to be saved and restored if necessary. The side effect of this code example lies in the LODS instruction that uses the EDI as destination pointer which gets incremented when the instruction is executed. By swapping the destination register and ESI before and after the LODS instruction, this code fragment actually increments the content of the destination register, thus emulating the INC instruction.

```
;Location of input values: EAX and EBX

    XOR EAX, EBX

      ⇓

    MOV ECX, 32
msb_a_to_CF:
    DEC ECX
    JZ end_of_cal
    RCL EAX,1
    JC a_is_one
    RCL EBX,1
    JMP msb_a_to_CF
a_is_one:
    RCL EBX,1
    JC CF_0
    JMP CF_1
CF_0:
    CLC
    JMP msb_a_to_CF
CF_1:
    STC
    JMP msb_a_to_CF
end_of_cal:

;Location of output value: EAX
```

Listing 1 – XOR using the carry flag.

```
;Location of input value: EAX

TEST EAX, EAX
JNZ SHORT 70

    ⇓

MOV ECX, EAX
INC ECX
LOOP 70
```

Listing 2 – Conditional jump with LOOP instruction.

```
;Location of input value: EAX

SUB EAX,100

    ⇓

MOV ECX,100
XCHG EAX,ECX
LOOP -1

;Location of output value: EAX
```

Listing 3 – SUB with LOOP instruction.

The functionality of ADD can be constructed in much the same way by loading the first operand into ESI and the second operand into ECX and executing the snippet above with the REP prefix. Each iteration adds 4 bytes to ESI, so the value of the repetition counter has to be set to a fourth of the value that should be added. Note that in the example in Listing 5 the value of EAX must point to a valid memory location and allow it to remain a valid memory location while iteratively increasing it to EAX+20. The same goes for the value of EDI. Thus, not all input values are valid, still, most will work. DEC and SUB can be emulated as well using the same set of instructions with the direction flag set (using the STD instruction).

Listing 6 shows another code fragment which uses side effects of a string instruction to hide arithmetic operations. In this example, the fact that the ESI is incremented by its size (32 bit) each time the MOVS instruction is executed, is used to generate a hidden SUB.

Again, like in the previous example EAX must point to a valid and readable memory location as this value is read by the MOVS instruction. Note that in Listings 4 to 6 the values of registers ECX, ESI, and EDI are modified. If these registers are used at this location, their values have to be saved and restored.

The complexity of automated and manual semantic analysis can be increased further by adding the REP prefix to LODS or dropping it for other string instructions.

For example, MOVS increments or decrements both ESI and EDI and is otherwise equivalent to MOV [EDI],[ESI].[1] Listing 7 shows an example of replacing MOV instructions with string instructions.

3.4. *Instruction set extensions*

Following the increasing demand for performant multimedia computing, CPU manufacturers have been adding new extensions to the original x86 instruction set. Starting with the x87 FPU (Floating Point Unit), a vast number of features such as Streaming SIMD Extensions (SSE, in its various versions up to 4.2), MMX and Advanced Vector Extensions (AVX) are present in current x86 CPUs. These extensions usually operate in

[1] This is a memory-to-memory move and therefore not a legal instruction as such; keeping this in mind, one can actually obfuscate moves between different registers or between registers and memory by storing them to memory first and then using MOVS without REP.

```
;Location of input value: EAX

INC EAX

    ⇓

XCHG EAX, ESI
LODS
XCHG EAX, ESI

;Location of output value: EAX
```

Listing 4 – Arithmetic operations with string instructions.

```
;Location of input value: EAX

ADD EAX, 80

    ⇓

MOV ESI, EAX
MOV ECX, 20
REP MOVS EDI, ESI
MOV EAX, ESI

;Location of output value: EAX
```

Listing 5 – Arithmetic operations (ADD) with string instructions.

a Single Instruction, Multiple Data (SIMD) fashion, i.e., a single instruction is applied to multiple input values. While SIMD instructions offer little advantage outside their domain (that is, multimedia or other vector operations), they can still be used to make reverse engineering attempts considerably harder. Real-life malware rarely employs these instructions, likely for lack of knowledge and economic reasons. Given the economic aspects of current malware, obscure features also reduce the potential number of installations (and therefore, profit). Due to these factors, most malware analysts and automated semantic-aware tools will not recognize these new instructions. One example of malware that employs MMX instructions is *W64/Sigrun*, which is also known as *W32/Svafa* (Ferrie, 2012). MMX, which was released in 1997, introduced eight new 64-bit registers and a number of new instructions. Interestingly, these registers are not *new* as such, but rather the existing eight 80-bit floating point registers (the most significant 16 bits are not used for MMX instructions, but will be clobbered by them). Later extensions added entirely new registers.

A comprehensive overview of all new instructions introduced throughout the past 15 years is outside the scope of this paper; however, it should be noted that a vast majority of them can be repurposed to obfuscate data movement or arithmetic instructions, for instance by including additional fake data in MMX/XMM registers or constructing special data for PXOR[2] and related boolean logic instructions.[3]

4. Compile-time obfuscation

The concept of Covert Computation works on a low level of abstraction, utilizing side effects in the hardware. Therefore, this type of code obfuscation is difficult to implement in high-level programming languages such as C, as the specific

[2] PXOR calculates the XOR of two MMX registers or an MMX register and memory or an immediate value.

[3] XOR is still one of the more popular basic operations in encryption employed by packers.

```
;Location of input value: EAX

SUB EAX, 80

    ⇓

MOV ESI, EAX
MOV ECX, 20
STD
REP MOVS EAX, EAX

;Location of output value: EAX
```

Listing 6 – Arithmetic operations (SUB) with string instructions.

implementation on binary level by the compiler is out of control of the developer. During the compilation process, the original code represented in some high-level language is converted to various intermediate representations (e.g., Register Transfer Language in the GCC compiler) and runs through several optimization cycles that make the code more efficient by, e.g., removing unnecessary instructions or converting complex code into simpler code (Madou et al., 2006). It would not be feasible to implement the obfuscation technique in a high-level representation, because the intended effects on the hardware would most likely get lost during the compilation process. On the other hand, injecting side effects in the binary code (*binary rewriting* (Smithson et al., 2010; Schwarz et al., 2002)) is error prone and complex, particularly when other obfuscation techniques are applied to the program in order to make disassembling harder. We therefore propose to apply code obfuscation at compile-time in order to benefit from both approaches while mitigating the discussed limitations. At compile-time all the required meta information (e.g., location information) is still present, allowing a more structured view on the code while the effects of compiler optimizations are controllable as the obfuscation modifications are applied in the same step. Thus, we consider compile-time obfuscation to be the only reasonable way of implementing covert instructions.

For our approach, we modified the compilation process of the LLVM (Low Level Virtual Machine) Compiler Infrastructure (Lattner and Adve, 2004) to insert the covert instructions directly into the hardware-specific assembly representation of the code. Fig. 2 shows the compilation process of LLVM as well as our modifications. We split this workflow into two parts: In our modified workflow, the program's code first runs through all optimization steps, the linker, and the target code generator and stops after the generation of hardware-dependent assembly code. At this point of the compilation process the code is already optimized for the specific target hardware, yet memory is still referenced by labels. Thus, modifications to the code can be performed without the need of rewriting jump target addresses, etc., reducing the complexity of the obfuscation process drastically. At this point, we apply our obfuscation method by first identifying functionality that can be implemented using side effects, then removing it from the code and finally injecting innocent looking code containing the same functionality inside side effects. After performing

```
;Location of input value: [ESI]

MOV EAX, [ESI]
MOV [EDI], EAX

    ⇓

MOVS EDI, ESI
MOV EAX, ESI

;Location of output value: EAX
```

Listing 7 – MOV with string instructions.

the obfuscating transformations, the second part simply takes the modified assembly code and converts it into an executable binary.

For our prototype implementation, we have written a compiler wrapper which can be used to easily integrate the concept of compile-time obfuscation into an existing toolchain (e.g., the GNU toolchain). The basic concept of the wrapper is that command line arguments that are passed to the wrapper are simply forwarded the actual compiler with one exception. Each command line argument that refers to a source code file (e.g., .c) is matched, the corresponding file is compiled to hardware-dependent assembly code and functionality is reimplemented using side effects. In our prototype implementation we used simple regular expressions for the identification of candidate functionality and reimplemented it with a semantically equivalent code block that hides the functionality in side effects. In a last step, the modified assembly file is passed, together with the other command line arguments, to the actual compiler, which finally generates the executable.

This approach makes it possible to insert covert instructions at compile time by just using our compiler wrapper instead of the actual compiler (e.g., specified in the Makefile of a software project) without requiring major modifications of the default toolchain. The original compiler is still used; however, instead of source code files, it receives modified assembly code.

5. Security analysis

In this section we discuss the effectiveness of our obfuscation technique and evaluate its impact on performance and binary size. We first considered assessing its resilience against commercial malware detectors by using real malware samples that were modified to implement some of their functionality in side effects. However, as pointed out by Moser et al. (2007), this type of evaluation would be of doubtful value. The detection engines of today's virus scanners are mainly signature-based, which means that modifying the binary code would most likely destroy the signature. It would then come as no surprise to have a detection rate that was lower than the one for the original binaries. As this effect can be simply tracked down to the modification of the signature and not to the concept of covert functionality in the code, it would heavily restrict the significance of the evaluation. Therefore, we decided to focus our evaluation on a theoretical analysis to evaluate the resilience of our approach against semantic-aware malware detection introduced by Christodorescu et al. (2005).

5.1. *Resilience against semantic-aware detection approaches*

For semantic-aware malware detection (Christodorescu et al., 2005), the binary program is disassembled and brought to an architecture-independent intermediate representation, which is then matched against templates describing malicious behavior. In order to be able to detect basic obfuscation methods like *register reassignment* or *instruction reordering* (e.g., by inserting jumps in the control flow graph), so-called def-use chains (relationship between the definition of a variable and the use of the same variable somewhere else in the program) are utilized. Furthermore, a value-preservation oracle is implemented for detecting NOP instructions and NOP fragments.

5.1.1. *IR Normalization*

The approach introduced by Christodorescu et al. (2005) is based on IDAPro for decompilation of the program to be analyzed. By generating an intermediate representation, semantically equivalent instruction replacements such as INC EAX, ADD EAX, 1, and SUB EAX, −1 are normalized with semantically disjoint operations and can then be matched against the generic template, which describes malicious behavior.

5.1.2. *Semantics detection*

Since the general problem of deciding whether one program is an obfuscated form of another program is closely related to the halting problem, which in general is undecidable (Turing, 1936), the presented algorithm uses the following strategy to match the program to the template: The algorithm tries to match (unify) each template node to a node in the program. In case two matching nodes are found, the def-use relationships in the template are evaluated with respect to the program code. If they hold true in the actual program, the program fragment matches the template.

5.1.3. *Value preservation and NOP detection*

The goal of this analysis step lies in the detection of NOP fragments, i.e., instruction sequences that do not change the values of the watched variables. The following strategies were implemented by Christodorescu et al. (2005): (i) Matching instructions against a library of known NOP instructions and NOP fragments, (ii) symbolic execution with randomized initial states, and (iii) two different theorem provers.

5.1.4. *Resilience against the approach*

As outlined by the authors, the semantic-aware malware detection approach is able to detect *instruction reordering* and *register reassignment* as well as a *garbage insertion*. Furthermore, with respect to the underlying instruction replacement engine, a limited set of *replaced instructions* can be detected. However, this approach is not able to detect obfuscation techniques using *equivalent functionality* or *reordered memory access*. In Fig. 3 we give an example of a code fragment (left) that is matched to a template (center) and an obfuscated form (right) of the same fragment. The obfuscation steps applied are flagged with the letters (A) and (B). Note that for reasons of simplicity, JMP instructions have been omitted from the illustration.

Since our obfuscation technique does not work by inserting NOP fragments, the direct detection and removal of them has no impact on our approach. Nevertheless, we use these mechanisms in the course of the matching algorithm in order to check for value preservation. The semantic detection relies heavily on the algorithm applying local unification by trying to find bindings of program nodes to template nodes. It is important to note that the bindings may differ at different program points, i.e., one variable in the template may be

bound to different registers in the program, and the binding is therefore not consistent. The idea behind this approach lies in the possibility to detect register reassignments. In order to eliminate inconsistent matches that cannot be solved using register reassignment, a mechanism based on def-use chains and value-preservation (using NOP detection) is applied.

The local unification used to generate the set of candidate matches that is then reduced using def-use chains and value preservation is limited by several restrictions. The following two are the most important ones with respect to our obfuscation method: (i) If operators are used in a template node, the node can only be unified with program nodes containing the same operators and (ii) symbolic constants in template nodes can only be unified with program constants. The obfuscation pattern (B) in Fig. 3 violates restrictions (i) and (ii) as, e.g., the simple "+"-function is replaced by a MOV instruction followed by looping an XCHG instruction. The same holds true for obfuscation pattern (C). In case of obfuscation pattern (A) even the control flow graph was changed as the explicit jump instruction following the condition as well as the condition itself are replaced by an assignment and a LOOP instruction. Thus, the local unification engine is not able to match these program fragments to the respective template fragments.

In order to generate the set of match candidates, the local unification procedure must be able to match program nodes with template nodes, relying on the IR-engine to detect semantically identical program nodes and to convert them into the same intermediate representation. However, authors state that "[…] same operation […] has to appear in the program for that node to match.". For example, an arithmetic left shift ($eax = eax \ll 1$) would not match a multiplication by 2 ($x = x * 2$) despite these instructions being semantically equivalent. Therefore, we can safely conclude that replacements with side effects as proposed in our concept would not match in the local unification as they do not use the same operations as the original code for implementing a specific functionality.

One could argue that once the concept of Covert Computation is publicly known, malware detectors could simply improve the hardware models on which the instruction replacement engine is based to be able to identify malicious behaviors implemented in side effects. While in theory, every single aspect of the hardware could be mapped to the machine model, we strongly believe that this is an unrealistic assumption for real-life applications. Increasing the level of detail and completeness of the model is costly in terms of analysis performance. Thus, its practical applicability in real-life malware detection scenarios, where the decision on maliciousness has to be made in real time, is limited. A more complex model also increases the complexity of the evaluation, so the model has to be kept as general as possible, preventing completeness in semantic-aware program analysis. Today's virus scanners as well as semantic-aware malware detection concepts are not even able to cover the entire semantics of code free from side effects. Following the original argument of the possibility of a complete model, mapping these semantics should have been even more trivial. Additionally, there is another crucial aspect that significantly limits detection. The model does not only have to be complete, it also has to be able to detect equivalence on a semantic level. However, based on Turing's halting problem (Turing, 1936), we know that deciding equivalence is not possible in general.

Another important aspect is diversity. Christodorescu et al. (2005) argue that a malware author would have to "devise multiple equivalent, yet distinct, implementations of the same computation, to evade detection". With Covert Computation we have shown that side effects in the microprocessor can be used to achieve exactly this requirement.

Table 2 – Impact on binary size.

Tool	Version	Size (normal)	Size (obfuscated)	%
MD5	2.2	12,101 bytes	12,227 bytes	1,04
bzip2	1.0.6	109,459 bytes	116,975 bytes	6,87
aescrypt	3.05	46,398 bytes	46,718 bytes	0,69

6. Evaluation

To evaluate the practicability of our approach we compared obfuscated binaries of three different Unix programs against their non-obfuscated versions. In addition, we performed a manual analysis of size and complexity overhead with real malware samples.

6.1. Prototype implementation

We measured performance and binary size overhead using our prototype implementation that intercepts the compilation process of LLVM (Lattner and Adve, 2004). We selected three Unix programs (MD5,[4] bzip2, aescrypt) and compiled each of them with two different configurations. The first version was compiled without any modifications to the code with LLVM, while the second one implements functionality inside side effects for the ADD and the SUB instruction.

6.1.1. Binary size

Binary size increase depends on the frequency of instructions that are replaced with semantically equivalent sequences of instructions. Table 2 shows a comparison of the binary size of the three programs used in our evaluation. Bzip2 contained a proportionally large number of ADD and SUB instructions, therefore, the increase of binary size was larger than for the other two programs. Still, we consider an increase of about 7%, which is well below similar approaches such as Wu et al. (2010), as acceptable.

6.1.2. Performance

For the performance evaluation, we ran all three programs on four different input files: an *audio* file (221 MB, Audio file with ID3 version 2.2.0, contains MPEG ADTS, layer III, v1, 192 kbps, 44.1 kHz, Stereo[5]), *plain text* (127.7 MB, UTF-8 Unicode English text, with very long lines), a *movie* (203.2 MB, ISO Media, Apple QuickTime movie), and an *image* (299.9 MB, TIFF image data, little-endian). The programs were run with default settings, for the evaluation of aescrypt we performed one entire encryption/decryption run. We measured the execution time

[4] http://www.fourmilab.ch/md5/.
[5] Output of the Unix file command.

for each program (normal and obfuscated) and calculated the arithmetic average of ten independent runs each. The results can be found in Fig. 4

In our prototype the implementation of functionality in side effects does not noticeably increase compilation time. The actual compilation process is by far the more time consuming task than the implementation of functionality in side effects. Thus, compilation time overhead is insignificant.

Except from MD5 the test runs with the obfuscated binaries show performance decreases well below 15%. The best results were achieved with aescrypt for which the biggest runtime increase we were able to measure was 3.1% for the TIFF file. The main reason for these differences is that the performance of code based on side effects for ADD and SUB instructions depends heavily on the operand that contains the value that is added to or subtracted from the specified register. A higher value requires the loop (refer to Listing 3) to be executed more often in order to generate the correct value in the target register. In the case of aescrypt, these values were considerably lower on average than in MD5 and bzip2.

6.2. Real malware samples

We further theoretically evaluated implications of Covert Computation on program complexity and size of the modified code based on tests with four recent malware samples we obtained from iSecLab's Anubis (iSeclab, 2009). In particular, we used samples of *Win32/Dorkbot*, *Win32/Gamarue*, *Win32/Sality-A*, and *Win32/Yeltminky* for our evaluation. We calculated the average complexity increases as well as the growth of the binary for three different implementation rates of Covert Computation. In this evaluation, 10, 20, and finally 50% of MOV, SUB, and INC instructions were replaced by covert code as described in Section 3. Fig. 5a shows the size overhead for the four tested malware samples in detail. In the first case, where 10% of all suitable MOV, SUB, and INC instructions were hidden inside side effects of innocent looking code, the size overhead for all four malware samples was below five percent, whereas for the highest implementation rate (50%), the overhead was between 14 and 23 percent. Dorkbot's large space overhead results from the high percentage of MOV instructions. Almost one third of this malware's instructions are MOVs.

As Fig. 5b shows, the complexity was heavily increased by the implementation of covert code sequences. For a 10% replacement rate, the execution complexity for the four malware samples was about 4.5 times the complexity of the unmodified versions of the code. We calculated a maximum of 8.89 for the malware sample of Dorkbot in case 50% of the instructions are replaced. While these increases in complexity, which cause performance slowdowns, are rather severe, we argue that certain types of malware are not performance critical. An example would be slow-spreading worms such as Code Red (Zou et al., 2002), which try to silently infect a large number of machines. The primary aim of this type of malware is to operate as stealthy as possible, while performance is of minor interest. Therefore, we strongly believe that there is a real threat of malware that implements hidden functionality in a trade-off with performance.

6.3. Limitations

A possible attack on the concept of Covert Computation is to statistically analyze the frequency of opcodes and compare them to samples of non-malicious programs. This idea of malware detection by analyzing opcode distribution was introduced by Bilar (2007). The paper concludes that the distribution of common opcodes is a relatively weak predictor for the maliciousness of software. In our evaluation, we came to a similar conclusion. As Table 3 shows, the four evaluated malware samples have very different opcode distribution patterns. While MOV instructions represent over 32% of all instructions of the malware sample *Dorkbot*, the code of *Sality* contains only about 25% MOV instructions. Replacing some of these instructions with semantically different sequences of instructions does not implicitly result in an uncommon and thus suspicious distribution of opcodes. However, as concluded by Bilar (2007), the frequency of rarely used opcodes is far more important for malware detection. Opcodes that are rarely used by compilers indicate additional optimizations and fine-tuning adjustments of the code – which is common for malware according to Bilar (2007). Some of the opcodes we use as hosts for hidden functionality, such as LOOP and RCL, are not commonly used by compilers and, therefore, overrepresented in programs that implement our approach. As a simple mitigation strategy, various host code fragments with different opcodes can be used on an alternating basis in order to keep the distribution of instructions as unobtrusive as possible.

Moreover, dynamic analysis techniques (such as evaluating the maliciousness of a program by monitoring system calls (Bayer et al., 2009)) are entirely unaffected by side-effect based obfuscation as they analyze the effects of the code rather than the code itself. However, in a malware context dynamic analysis requires the evaluated program to be run in a protected environment in order to prevent harmful actions to the host it is run. Thus, in host based malware analysis scenarios (end-user virus scanners), dynamic analysis does not play a major role.

7. Conclusions and future work

In this paper we proposed the obfuscation concept Covert Computation which hides (malicious) code in innocent looking programs. We have shown that the complexity of today's microprocessors, which support a large set of different instructions, can be exploited to hide functionality in a program's code as small code portions. Our prototype

Table 3 – Opcode frequency in malware samples (D = Dorkbot, G = Gamarue, S = Sality, Y = Yeltminky).

	D	%	G	%	S	%	Y	%
MOV	611	32.40	207	25.91	79	21.41	126	29.65
SUB	57	3.02	117	14.64	3	0.81	1	0.24
INC	42	2.23	43	5.38	22	5.96	8	1.88
AND	32	5.24	14	6.76	3	3.80	0	0
XOR	79	4.19	38	4.76	1	0.27	3	0.71

implementation is based on the idea of compile-time obfuscation that allows to apply code obfuscation during compilation in order to mitigate problems resulting from applying the obfuscating transformation either too early at source code level (before code optimization at compile time) or in the final binary (where it is difficult to validate the correctness of modifications and a lot of meta-data needed for efficient obfuscation is missing). With the help of a prototype implementation, which perfectly integrates into existing software development toolchains, we were able to show the practicability of our approach. With moderate overhead, it is possible to hide possibly malicious functionality in a program's code.

In our future research, we aim to tackle the discussed limitation of opcode distribution by combining our approach with the concept of *mimimorphism* (Wu et al., 2010) in order to generate code with a distribution of opcodes that mimics any non-malicious reference program. Furthermore, we aim at extending our concept to other microprocessor architectures. RISC based platforms such as ARM are of particular interest as their less-complex instructions are less likely to include side effects, thus it is more difficult to implement covert functionality.

Acknowledgments

The research was funded under Grant 826461 (FIT-IT) by the FFG – Austrian Research Promotion Agency and the Josef Ressel Center for User-friendly Secure Mobile Environments by the Christian Doppler Research Association (Project 627).

REFERENCES

Bayer U, Habibi I, Balzarotti D, Kirda E, Kruegel C. A view on current malware behaviors. In: Proceedings of the 2nd USENIX conference on large-scale exploits and emergent threats: botnets, spyware, worms, and more. USENIX Association; 2009 [pp. 8–8].

Bilar D. Opcodes as predictor for malware. Int J Electron Secur Digital Forensics 2007;1(2):156–68.

Bruschi D, Martignoni L, Monga M. Detecting self-mutating malware using control-flow graph matching. In: Detection of intrusions and malware & vulnerability assessment 2006. pp. 129–43.

Chow S, Eisen P, Johnson H, Van Oorschot P. White-box cryptography and an aes implementation. In: Selected areas in cryptography. Springer; 2003. pp. 250–70.

Christodorescu M, Jha S. Testing malware detectors. ACM SIGSOFT Softw Eng Notes 2004;29(4):34–44.

Christodorescu M, Jha S, Seshia S, Song D, Bryant R. Semantics-aware malware detection. In: Security and privacy, 2005 IEEE symposium on. IEEE; 2005. pp. 32–46.

Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations [Tech. rep]. New Zealand: Department of Computer Science, The University of Auckland; 1997.

Crandall J, Su Z, Wu S, Chong F. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proceedings of the 12th ACM conference on computer and communications security. ACM; 2005. pp. 235–48.

Dalla Preda M, Christodorescu M, Jha S, Debray S. A semantics-based approach to malware detectionIn ACM SIGPLAN Notices, vol. 42. ACM; 2007. pp. 377–88.

Dalla Preda M, Christodorescu M, Jha S, Debray S. A semantics-based approach to malware detection. ACM Transact Program Lang Syst (TOPLAS) 2008;30(5):25:1–25:53.

De Sutter B, Anckaert B, Geiregat J, Chanet D, De Bosschere K. Instruction set limitation in support of software diversity. In: Information security and cryptology–ICISC 2008 2009. pp. 152–65.

Egele M, Scholte T, Kirda E, Kruegel C. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput Surv (CSUR) 2012;44(2):6.

Ferrie P. This sig doesn't run. Virus Bulletin; January 2012.

Giacobazzi R. Hiding information in completeness holes: new perspectives in code obfuscation and watermarking. In: Software engineering and formal methods, 2008. SEFM'08Sixth IEEE International Conference on. IEEE; 2008. pp. 7–18.

Griffin K, Schneider S, Hu X, Chiueh T. Automatic generation of string signatures for malware detection. In: Recent advances in intrusion detection. Springer; 2009. pp. 101–20.

iSeclab. Anubis [Online; retrieved January 12th, 2013], http://anubis.iseclab.org; 2009.

Kinder J, Katzenbeisser S, Schallhart C, Veith H. Detecting malicious code by model checking. In: Detection of intrusions and malware, and vulnerability assessment 2005. pp. 514–5.

Kolbitsch C, Comparetti P, Kruegel C, Kirda E, Zhou X, Wang X. Effective and efficient malware detection at the end host. In: Proceedings of the 18th conference on USENIX security symposium. USENIX Association; 2009. pp. 351–66.

Lanzi A, Balzarotti D, Kruegel C, Christodorescu M, Kirda E. Accessminer: using system-centric models for malware protection. In: Proceedings of the 17th ACM conference on computer and communications security. ACM; 2010. pp. 399–412.

Lattner C, Adve V. Llvm: a compilation framework for lifelong program analysis & transformation. In: Code generation and optimization, 2004. CGO 2004. International symposium on. IEEE; 2004. pp. 75–86.

Lyda R, Hamrock J. Using entropy analysis to find encrypted and packed malware. Secur Priv IEEE 2007;5(2):40–5.

Madou M, Anckaert B, De Bus B, De Bosschere K, Cappaert J, Preneel B. On the effectiveness of source code transformations for binary obfuscation; 2006.

Moser A, Kruegel C, Kirda E. Limits of static analysis for malware detection. In: Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual. IEEE; 2007. pp. 421–30.

Nachenberg C. Computer virus-coevolution. Commun ACM 1997;50(1):46–51.

O'Kane P, Sezer S, McLaughlin K. Obfuscation: the hidden malware. Secur Priv IEEE 2011;9(5):41–7.

Schrittwieser S, Katzenbeisser S, Kieseberg P, Huber M, Leithner M, Mulazzani M, et al. Covert computation: hiding code in code. In: In Proceedings of the 8th ACM symposium on information, computer and communications security (ASIACCS 2013). ACM; 2013.

Schwarz B, Debray S, Andrews G. Disassembly of executable code revisited. In: Reverse engineering, 2002. Proceedings. Ninth working conference on. IEEE; 2002. pp. 45–54.

Sharif M, Yegneswaran V, Saidi H, Porras P, Lee W. Eureka: a framework for enabling static malware analysis. In: Computer security-ESORICS 2008 2008. pp. 481–500.

Smithson M, Anand K, Kotha A, Elwazeer K, Giles N, Barua R. Binary rewriting without relocation information. USPTO patent pending no. 12 785. 2010.

Song Y, Locasto ME, Stavrou A, Keromytis AD, Stolfo SJ. On the infeasibility of modeling polymorphic shellcode. In:

Proceedings of the 14th ACM conference on computer and communications security. ACM; 2007. pp. 541–51.
Turing A. On computable numbers, with an application to the entscheidungsproblem. In: Proceedings of the London mathematical society, vol. 42; 1936. pp. 230–65.
Weaver R. A probabilistic population study of the conficker-c botnet. In: Passive and active measurement. Springer; 2010. pp. 181–90.
Willems C, Holz T, Freiling F. Toward automated dynamic malware analysis using cwsandbox. IEEE Secur Priv 2007:32–9.
Wu Z, Gianvecchio S, Xie M, Wang H. Mimimorphism: a new approach to binary code obfuscation. In: Proceedings of the 17th ACM conference on computer and communications security. ACM; 2010. pp. 536–46.
Zou C, Gong W, Towsley D. Code red worm propagation modeling and analysis. In: Proceedings of the 9th ACM conference on computer and communications security. ACM; 2002. pp. 138–47.

Sebastian Schrittwieser is a PhD candidate at the Vienna University of Technology and a researcher at SBA Research, the Austrian non-profit research institute for IT-Security. His research interests include, among others, digital forensics, software protection, code obfuscation, and digital fingerprinting. Sebastian received a Dipl.-Ing. (equivalent to MSc) degree in Business Informatics with focus on IT security from the Vienna University of Technology in 2010.

Stefan Katzenbeisser is a full professor at TU Darmstadt, where he is heading the Security Engineering Lab (SecEng). His main research interests are in the area of the design and analysis of cryptographic protocols, privacy-enhancing technologies, and software security.

Peter Kieseberg is a researcher at SBA Research, the Austrian non-profit research institute for IT-Security. He received a Dipl. Ing. (equivalent to MSc) degree in Technical Mathematics in Computer Science from the Vienna University of Technology. His research interests include digital forensics, fingerprinting of structured data and mobile security.

Markus Huber is a computer security and privacy researcher from Austria. He works for SBA Research, an industrial research center for IT-Security founded by the Vienna University of Technology, Graz University of Technology, and University of Vienna.

Manuel Leithner is researcher at SBA Research, the Austrian non-profit research institute for IT-Security. His research interests include mobile security, cloud computing and code obfuscation.

Martin Mulazzani is a Ph.D. student in Computer Science at the Vienna University of Technology, Vienna, Austria; and a Computer Security Researcher at SBA Research, Vienna, Austria. His research interests include privacy, digital forensics and applied security.

Edgar Weippl is the Research Director at SBA Research, Vienna, Austria; and an Associate Professor of Computer Science at the Vienna University of Technology, Vienna, Austria. His research focuses on information security and e-learning.