Covert Computation
Hiding Code in Code for Obfuscation Purposes

Sebastian Schrittwieser*, Stefan Katzenbeisser‡, Peter Kieseberg†, Markus Huber†,
Manuel Leithner†, Martin Mulazzani†, Edgar Weippl†

Vienna University of Technology*
sebastian.schrittwieser@tuwien.ac.at

Darmstadt University of Technology‡
skatzenbeisser@acm.org

SBA Research†
{pkieseberg, mhuber, mleithner, mmulazzani, eweippl}@sba-research.org

ABSTRACT

As malicious software gets increasingly sophisticated and resilient to detection, new concepts for the identification of malicious behavior are developed by academia and industry alike. While today's malware detectors primarily focus on syntactical analysis (i.e., signatures of malware samples), the concept of semantic-aware malware detection has recently been proposed. Here, the classification is based on models that represent the underlying machine and map the effects of instructions on the hardware. In this paper, we demonstrate the incompleteness of these models and highlight the threat of malware, which exploits the gap between model and machine to stay undetectable. To this end, we introduce a novel concept we call *covert computation*, which implements functionality in side effects of microprocessors. For instance, the flags register can be used to calculate basic arithmetical and logical operations. Our paper shows how this technique could be used by malware authors to hide malicious code in a harmless-looking program. Furthermore, we demonstrate the resilience of *covert computation* against semantic-aware malware scanners.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General-Security and protection

Keywords

code obfuscation; side effects; malware detection

1. INTRODUCTION

Malware detection is an important research problem in computer security that strives to spot malicious routines in software. In recent years, the threat of malware, viruses, spyware, and trojans has dramatically increased and resulted in a cat-and-mouse game between malware authors and developers of anti-malware software. The ultimate goal of malware detectors (commonly known as virus scanners) is to determine whether a program includes malicious routines or not. Since the early days of malware defense, this was done by matching a signature of known malware against the software to be analyzed [2]. With the increasing amount of malicious software, the extraction of signatures from malware samples as the sole detection technique became inefficient as well as insufficient. Over time, other identification methods were developed and used in combination with malware signatures, which are still widely used by anti-malware software [9]. Heuristic-based malware detection identifies malicious code by statistically analyzing its structure and behavior without depending on prior knowledge of the malware. However, this approach suffers from false positives as well as false negatives as decisions are based on statistical models for maliciousness. Furthermore, the rise of concepts such as polymorphism and metamorphism lead to an entirely new class of malware, which is resistant against signature-based detectors, as these focus exclusively on malware syntax and ignore malware semantics. The idea of semantic-aware malware detection was introduced by Christodorescu et al. in 2005 [3]. Their approach is based on the definition of templates for malicious behavior and is more resistant to simple obfuscation techniques such as *garbage insertions* [5] and *equivalent instruction replacement* [7] as the semantics of the code are analyzed.

In this paper, we demonstrate that today's static malware detection approaches ignore fundamental knowledge of the underlying hardware and thus are ineffective against our novel *covert computation* obfuscation method. Static analysis techniques for binary code are based on a specific machine model in order to understand the functionality of the analyzed program. Such a model describes how code is interpreted by the machine, i.e., how a specific instruction influences the state of the microprocessor. Based on the entirety of effects that a sequence of instructions has on the model, its maliciousness is evaluated by the malware detection software. This model, however, is a simplified, abstract representation of the real machine. This simplification poses a problem for model-based code analysis, as abstract models are not strong enough to entirely simulate the effects of

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.

the code running on real hardware. It is possible to refine machine models and make them more expressive, but this results in a tradeoff between correctness and complexity. In a malware detection context, the complexity of testing whether a given code matches a model for malicious behavior has to be low enough for the problem to be decidable in real time. The fundamental dilemma of static malware detection is that, on the one hand, code can be made arbitrarily complex with acceptable performance losses, while, on the other hand, a model strong enough to perform a complete evaluation of code semantics would reach an impractical level of complexity for real-life applications.

As main contributions of this paper we introduce a novel approach for code obfuscation called *covert computation*, based upon side effects in today's microprocessor architectures. We further show feasibility of our concept based on instruction side effects in the flags register as well as `LOOP` and string instructions. We finally demonstrate how our approach fundamentally raises the bar for semantic-aware code analysis.

2. RELATED WORK

The use of code obfuscation to prevent reverse engineering of any given software is a well-studied field [4, 17]. A formal concept of code obfuscation has been defined by Barak et al. [1]. Although this work shows that a universal obfuscator for any type of software does not exist and perfectly secure software obfuscation is not possible, various types of code obfuscation are still used by today's malware to "raise the bar" for detection.

Various malware obfuscation approaches presented in the literature follow the concept of polymorphism [13], which hides malicious code by packing or encrypting it as data that cannot be interpreted by the analysis machine. Thus, an unpacking routine has to be used to turn this data back into machine-interpretable code. A number of approaches have been suggested to defeat this obfuscation technique, such as detecting malicious code with model checking [11] or symbolic execution [6]. Today's malware detection systems evaluate the maliciousness of a program based on structural and behavioral patterns [10]. Christodorescu et al. [3] first introduced the concept of semantic-aware malware detection. In their paper, the authors define formal semantics for the maliciousness of programs and a semantic-aware matching algorithm for malware detection based on them. Templates for malicious behavior are defined and matched against the potential malware. If both have the same effect on memory, the binary is identified as malicious. This approach can deal with simple forms of obfuscation but does not recognize instruction replacements based on patterns completely. The concept was further formalized by Preda et al. in 2007 [14] and 2008 [15].

Recently, Wu et al. [18] introduced the concept of mimimorphism, which aims at obfuscating malicious code against both static and statistical detection systems. In the area of code obfuscation, various approaches for hiding a program's semantics can be found. In recent literature on code obfuscation, several authors have proposed the removal of instruction patterns in order to increase de-obfuscation complexity. Recent work by De Sutter et al. [7] on avoiding characteristic instruction patterns normalizes the distribution of instructions used in a program by replacing rare ones with semantically equivalent blocks of more frequently used instructions with equivalent blocks of less frequently used instructions. The drawback to this approach is, however, that an equal distribution of instructions used by a program is statistically unlikely and therefore easily detectable for a code analyst. Furthermore, semantic-aware detection approaches such as described in [3] can implement the replacement patterns in their templates for malicious behavior.

Giacobazzi [8] first theoretically discussed the idea of making code analysis more difficult by forcing the detection system to become incomplete. However, no practical approach of this idea was given in the paper. Moser et al. [12] discussed the question whether static analysis alone allows reliable malware detection.

3. APPROACH

All semantic-aware malware identification techniques follow the same basic approach. The classification of maliciousness is based on a model of the underlying system (i.e., the microprocessor), which describes how a specific instruction modifies the system's state. The quality and completeness of the model are crucial for a high identification rate. An incomplete model is not able to map all effects that an instruction has on the hardware and thus cannot evaluate its impact on the system and the system's state after executing the instruction. Current models for malware detection are focused on the instruction layer but do not fully map all effects of an instruction on the model. Today's microprocessors are highly complex systems with hundreds of different instructions that influence the processor's state.

In general terms, there are two types of models for us to consider. First, a human analyst defines his or her own model of the machine when trying to understand the meaning of a program's code. Given that the analyst knows the purpose of a specific instruction, he or she can perceive the code's meaning on a semantic layer and draw conclusions concerning the functionality of a sequence of instructions. However, it is drastically more difficult to keep the program's entire state, which is modified constantly while executing instructions, in mind. Thus, the human model of a machine can be described as a very basic semantic representation. The second model that has to be considered is the one of an automatic analysis tool, which makes a decision regarding the maliciousness of a program based on predefined templates and patterns. If side effects are not implemented in this model, its impact cannot be evaluated and is missed by the analysis tool.

In order to hide the implementation of a specific functionality of a program, we identified the possibility of implementing it based on features of the processor that are not described by its model and thus not evaluated. Analyzing such code on the semantic layer would not identify the hidden functionality as it is not contained in the basic semantics of a sequence of instructions, but in some deeper abstraction layer that is not included in the model.

3.1 Side Effects

In computer science, side effects in general describe any persistent modification of a program's state or its environment after executing a basic block, which is a straight block of instructions with one entry and one exit point. This includes setting a global variable, writing to the file system or accessing auxiliary equipment. Side effects are a fundamental prerequisite for a program's ability to interact with

the user, the underlying computing system or other programs. Depending on the programming paradigm, side effects are more or less frequently used in programming languages. While in functional programming languages, such as Haskell, side effects do not exist or are restricted to a minimum, imperative programming (e.g., C, Java, and many others) makes use of side effects more frequently.

While most side effects are intended by the developer and are an integral part of the functionality (e.g., writing data to the file system or interacting with the outside world over a network connection), some side effects modify states of the program or the outside world without the developer's direct knowledge. The developer has a mental model of instructions and its effects, which might not cover the entire functionality of the instruction. The same applies to a code analyst and even, in a similar form, to machine-based malware detection systems. Code is analyzed based on expectations of what functionality a specific instruction implements. This can be the mental model of the human code analyst or the hardware model that is implemented in the automatic malware analysis system. Consider the x86 instruction `ADD EAX, EBX`. The core functionality, which is expected by the analyst, is the calculation of the sum of the two operands `EAX` and `EBX`. Thus, the new state of the machine after the execution of the instruction includes a modified register `EAX` that now holds the sum of the registers `EAX` and `EBX`. However, there is another register that was influenced by the `ADD` instruction. Within the flags register, several bits could have been modified by the instruction, depending on the result of the operation. For example, the Zero flag is set to 1 if the result of the operation is 0.

3.2 Using side effects to hide functionality

Usually, side effects are avoided at an early stage of software development (e.g., developers are discouraged from using global variables heavily), as they could influence the program's state in a way that was not considered by the developer and cause an unpredictable malfunction of the program.

In a security context, we have identified side effects as excellent vehicles for hiding malicious functionality inside arbitrary program code. The idea of injecting malicious functionality by making program code look harmless is not new. Winning examples of the *Underhanded C Contest*[1], an annual contest for writing innocent-looking C code implementing covert malicious behavior, use very subtle techniques for hosting hidden functionality. In contrast to that contest, which is based on the high-level language C, we describe covert functionality that is tied very closely to the underlying hardware, based on side effects in the microprocessor and implemented at assembly level.

3.3 Flags

Flags in microprocessors are status bits that are used for indicating states and conditions of different operations performed by the microprocessor. For example, most of today's architectures such as x86 implement a Zero flag, which is set when the result of an arithmetic or logical operation is equal to zero. The value of a flag bit is then used for conditional jumps and is therefore responsible for modifying the control flow of a program's execution.

[1]http://underhanded.xcott.com (Last accessed February 12th 2013)

In the context of side effects within programs, flags are the target of side effects. In x86 as well as in many other microprocessor architectures, most arithmetic and logical operations influence at least one bit in the flags register. Flags can therefore be seen as global variables that are permanently modified by instructions. The concept of *covert computation* uses flags for basic operations in such a way that the flags are directly used for performing the calculation and storing intermediate results by using conditional jumps. In the following, we show how flags can be used as intermediate storage for the calculation of logical operations. This concept can easily be adopted for arithmetic operations.

In a bitwise logical operation the calculation of the four different possible combinations (00/01/10/11) can be represented by two conditional jumps. The two input operands are stored in two arbitrary flags. In a first step, the value of the first flag is determined by implementing a conditional jump, which evaluates this specific flag. In x86, for example, the Zero flag can be evaluated with a `JZ` (jump if zero) or a `JNZ` (jump if not zero) instruction. For each case (jump was performed or not performed), the second phase again implements a conditional jump, thus resulting in four possible outgoing control flows, each representing one possible result of the bitwise operation of the values stored in the two flags. Figure 1 illustrates the approach. The two source values 1 and 0 are stored in the Zero and the Carry flag. Two conditional jumps are implemented to calculate the logical `XOR`. The first one (`JZ`) evaluates the value stored in the Zero flag. For both cases – the jump is either performed or not – a second conditional jump (`JC`), which evaluates the carry flag, is implemented. The control flow of the software now follows the calculation of a logical XOR, and at each of the four possible end points of the control flow graph, code can be implemented that stores the result of the calculation to some output register.

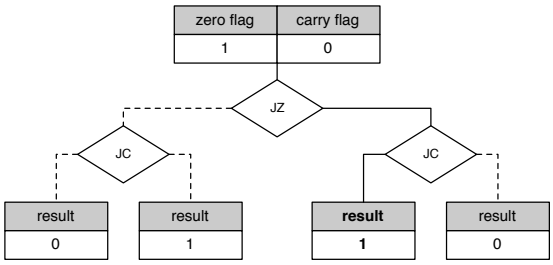

Figure 1: Calculation of XOR using the Zero and the Carry flag.

This concept can be extended to the full length of 32-bit operands very easily by repeating the two jumps for each bitwise operation. Other logical operations can be represented as side effects analogous to the `XOR` replacement pattern.

3.4 Other side effects in the x86 architecture

Many instructions of the x86 architecture modify the state of the processor in a way that is not the primary functionality of the instruction. In the following, we exemplarily describe side effects of the `LOOP` as well as string instructions and explain how side effects can be used to emulate other instructions.

LOOP instruction. The LOOP instruction in x86 behaves as follows: The value of the counter register (CX/ECX) is decremented by one. If it contains 0 after this operation, the loop terminates and execution continues past the LOOP instruction. Otherwise, a short jump to the relative offset specified as the operand to LOOP is taken. While the obvious behavior of the LOOP instruction is the repeated execution of one or more instructions, it also can be perform other, not so obvious functionality. One of the most trivial ways to repurpose this instruction's behavior is by using it as a short JMP or conditional jump. This can be achieved by ensuring that ECX does not contain 0 (or writing a value unequal to 0 to ECX if the given condition is met) and instead of using a JMP instruction, writing LOOP <label>. However, there are far more sophisticated ways to emulate other given instructions using LOOP. In Listing 1, the standard SUB instruction is represented using a combination of LOOP and XCHG.

```
SUB EAX,200

   ⇓

MOV ECX,200
XCHG EAX,ECX
LOOP -1
```

Listing 1: SUB with LOOP instruction

At first glance, this loop may seem entirely pointless, simply exchanging the values of EAX and ECX at every iteration and decreasing the value of ECX. However, because ECX constantly witches between containing the actual loop counter and the value of EAX, the latter is actually decremented by the former until either of them reaches the value 0. Note that this sequence of commands will not work as intended if EAX contains a value between 0 and 0x200. A variety of other instructions can be emulated using LOOP, e.g., MOV between registers (by moving the target value into ECX and incrementing the target register's value at every loop iteration).

String instructions. Another source of side effects in the x86 architecture are the string instructions. MOVS, SCAS, CMPS, STOS, and LODS instructions are intended to operate on continuous blocks of memory instead of single bytes. Because of the fact that these instructions modify the registers ESI, EDI, and ECX it is possible to emulate ADD, SUB, INC, and DEC instruction. An example of a replacement pattern for the INC instruction is given in Listing 2. The replacement pattern will clobber the value of ESI and is only applicable if the value of EAX points to a memory location that is accessible to the program.

```
INC EAX

   ⇓

XCHG EAX, ESI
LODS
XCHG EAX, ESI
```

Listing 2: Arithmetic operations with string instructions.

4. EVALUATION

In this section we discuss the effectiveness of the concept of *covert computation* for obfuscation purposes. 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 in [12], 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 the concept of semantic-aware malware detection. We performed a theoretical analysis to evaluate the resilience of our approach against Christodorescu et al.'s [3] approach. For semantic-aware malware detection, the binary program is disassembled and brought to an architecture-independent intermediate representation, which is 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 are utilized. Furthermore, a value-preservation oracle is implemented for detecting NOP instructions as well as NOP fragments.

Normalization of intermediate representation. The approach introduced by Christodorescu et al. [3] is based on IDAPro for decompilation of the program to be analyzed. By generating an intermediate representation (IR), semantically equivalent instruction replacement patterns 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.

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 [16], 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.

Value preservation and NOP detection. The goal of this analysis step lies in the detection of NOP operations, i.e., program fragments that do not change the values of the watched variables. The following strategies were implemented by the authors in [3]: (i) Matching instructions against a library of known NOP commands and NOP fragments, (ii) symbolic execution of the code sequence with randomized initial states, as well as (iii) two different theorem provers.

Resilience against the approach. As outlined by the authors, the semantic-aware malware detection approach is

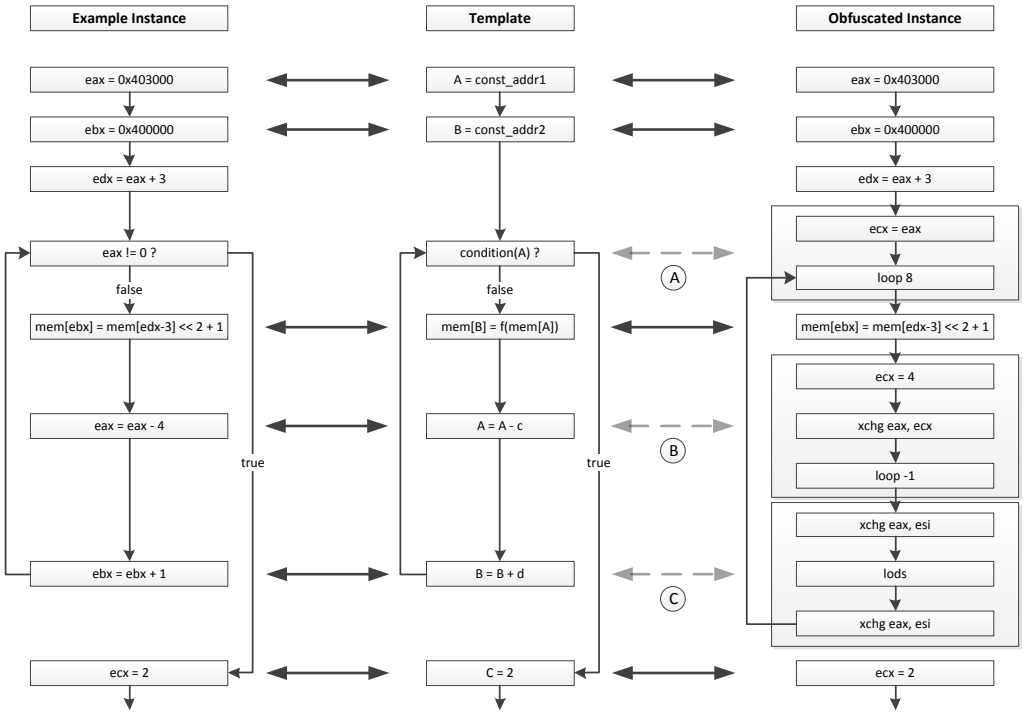

Figure 2: Resilience against semantic-aware malware detection.

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 Figure 2 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 `NOP` elements 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 Figure 2 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 << 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 as increasing the level of detail and completeness of the model is costly and reduces its practical applicability

in real-life malware detection scenarios, where the decision on maliciousness has to be made in real time. 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 side effects-free code. Following the original argument of the possibility of a complete model, mapping these semantics should have been even more trivial. The second important aspect is diversity. Christodorescu et al. [3] 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.

5. CONCLUSION

In this paper, we have shown that the complexity of today's microprocessors, which support a myriad of different instructions, can be exploited to hide functionality in a program's code as small code portions. In the context of malware, we demonstrated that existing concepts for semantic-aware malware detection systems are not able to analyze the semantics of code that is implemented in side effects and argue that the indirectly shown incompleteness of machine models for semantic-aware malware detection raises the threat of malware that exploits exactly this knowledge gap. A successful implementation of the obvious mitigation strategy – improving the models – is doubtful, as, while it might be possible to map the side effects of a specific instruction in the model, it would complex to evaluate the impact of the side effects of an entire sequence of instructions. Models are abstract representations of the real world, which implies that some information is lost in its development. The increasing complexity of the real world makes it hard to design models that are strong enough to entirely simulate the effects of the code running on real hardware. The obfuscator has an important advantage over code analysts as he or she can make the code arbitrarily complex. In contrast, analysts have to keep their models simple to avoid an impractical level of complexity for testing whether a given code matches a model for malicious behavior. Thus, the feasibility of implementing a complete hardware model for malware detection is an unrealistic assumption.

Acknowledgments

The research was funded under Grant 826461 (FIT-IT) and COMET K1 by the FFG – Austrian Research Promotion Agency.

6. REFERENCES

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology—Crypto 2001*, pages 1–18. Springer, 2001.

[2] M. Christodorescu and S. Jha. Testing malware detectors. *ACM SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004.

[3] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[4] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *Software Engineering, IEEE Transactions on*, 28(8):735–746, 2002.

[5] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[6] J. Crandall, Z. Su, S. Wu, and F. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248. ACM, 2005.

[7] B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. De Bosschere. Instruction set limitation in support of software diversity. *Information Security and Cryptology–ICISC 2008*, pages 152–165, 2009.

[8] R. Giacobazzi. Hiding information in completeness holes: New perspectives in code obfuscation and watermarking. In *Software Engineering and Formal Methods, 2008. SEFM'08. Sixth IEEE International Conference on*, pages 7–18. IEEE, 2008.

[9] K. Griffin, S. Schneider, X. Hu, and T. Chiueh. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection*, pages 101–120. Springer, 2009.

[10] N. Idika and A. Mathur. A survey of malware detection techniques. *Purdue University*, page 48, 2007.

[11] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 514–515, 2005.

[12] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430. IEEE, 2007.

[13] C. Nachenberg. Computer virus-coevolution. *Communications of the ACM*, 50(1):46–51, 1997.

[14] M. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *ACM SIGPLAN Notices*, volume 42, pages 377–388. ACM, 2007.

[15] M. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):25:1–25:53, 2008.

[16] A. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1936.

[17] S. Udupa, S. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[18] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang. Mimimorphism: a new approach to binary code obfuscation. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 536–546. ACM, 2010.