FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Hardware-assisted Code Obfuscation

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Sebastian Schrittwieser
Matrikelnummer 0325695

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: PD Dr. Edgar Weippl, Prof. Dr. Stefan Katzenbeisser

Diese Dissertation haben begutachtet:

(PD Dr. Edgar Weippl)

(Prof. Dr. Stefan Katzenbeisser)

Wien, 30.04.2014

(Sebastian Schrittwieser)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Hardware-assisted Code Obfuscation

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Sebastian Schrittwieser
Registration Number 0325695

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: PD Dr. Edgar Weippl, Prof. Dr. Stefan Katzenbeisser

The dissertation has been reviewed by:

(PD Dr. Edgar Weippl)

(Prof. Dr. Stefan Katzenbeisser)

Wien, 30.04.2014

(Sebastian Schrittwieser)

Erklärung zur Verfassung der Arbeit

Sebastian Schrittwieser
Lammgasse 6/13, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

It would not have been possible to write this dissertation without the support and help of all the kind people around me. First and foremost, I would like to express my deepest gratitude to my advisor Stefan Katzenbeisser for his excellent guidance, patience and the long discussions that helped me sort out the technical details of my work — even across large geographic distance. The same goes for Edgar R. Weippl, who provided me with an excellent atmosphere for doing research and the great freedom to pursue independent work.

Further, I would like to thank my great colleagues at SBA Research, especially Markus Huber, Martin Mulazzani, Manuel Leithner, Peter Kieseberg, and Peter Frühwirt for their insights, support of my research, and their friendship. Additionally, I am very thankful for the opportunity A Min Tjoa and his entire Information & Software Engineering Group gave me to write my dissertation at Vienna University of Technology. My gratitude also goes to the Austrian Research Promotion Agency (FFG), which provided funding for my dissertation (FIT-IT project No. 826461).

Above all, I am grateful for the support from my parents Gabriele and Karl, who motivated me to pursue my love for computer science and supported me throughout the years. For their careful proofreading, as well as their comments and suggestions, I would like to thank my brothers Florian and Julian. Finally, a very special acknowledgement goes to my girlfriend Alina for her endless love and encouragement throughout the writing phase of this dissertation. You always make me feel like anything is possible.

Abstract

Software obfuscation is a longstanding and open research challenge in computer security. While theoretical results indicate that provably secure obfuscation in general is impossible to achieve, many application areas (e.g. malware, commercial software, etc.) show that software obfuscation is indeed employed in practice. Still, it remains largely unexplored to what extent today's software obfuscation state-of-the-art can keep up with the progress in code analysis and where we stand in the arms race between attackers and defenders. The first part of this thesis thus analyzes how effective software obfuscation is in the presence of ever more sophisticated deobfuscation techniques and off-the-shelf code analysis tools. To this end, we develop a novel classification scheme for the resilience of different types of obfuscations in specific attack scenarios. The answer heavily depends on the goals of the attacker and his available resources. Even simple obfuscation techniques can be quite effective against analysis techniques employing pattern matching or static analysis, which explains the unbroken popularity of obfuscation among malware writers. Dynamic analysis methods, in particular if assisted by a human analyst, are much harder to cope with; this makes software obfuscation for the purpose of intellectual property protection highly challenging.

The subsequent part of this thesis therefore concentrates on code obfuscation for the protection of intellectual property in software. Software diversification is an effective method for preventing that automated attacks developed against one instance of a program work against other instances (*class break*). However, distribution of diversified software is challenging as each copy has to be different. The second research problem we consider in this thesis is the development of a concept for software diversification which does not require the individual copies of the program to be different on the binary code layer and thus provides a solution to the distribution problem. We introduce a novel code obfuscation scheme that applies the concept of software diversification to the control flow graph of the software. Our approach makes dynamic reverse-engineering considerably harder as the information an attacker can retrieve from the analysis of a single run of the program with a certain input is useless for understanding the program's behavior on other inputs.

While the resilience of code obfuscation remains unclear and ultimately depends only on available resources and patience of the attacker, hardware-based solutions (*trusted computing*) provide a wide range of protection mechanisms such as remote attestation and secure storage for secrets. However, until now almost no systematic research has been done on the interplay between hardware- and software-based protection mechanisms. The third research problem we tackle in this thesis is how code obfuscation can be assisted by lightweight hardware. We propose

minimal modifications to Intel's AES-NI instruction set in order to make it suitable for application in software protection scenarios and use these modifications for parametrization of our control flow obfuscation scheme. The combined approach provides strong hardware-software binding and restricts the attack context to pure dynamic analysis — two major limiting factors of reverse-engineering.

In the final part of this thesis, we focus on the problem of malware obfuscation. 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 and thus does not depend on a specific syntactic implementation. For this purpose a model of the underlying machine is used. The fourth research problem we deal with in this thesis is the implementation of hidden functionality based on properties that are difficult to cover with a model of the hardware. We present COVERT COMPUTATION, a concept for the implementation of functionality in side effects of the microprocessor. We further give a comprehensive analysis of side effects in the x86 architecture and demonstrate the suitability of COVERT COMPUTATION for malware obfuscation.

Kurzfassung

Software-Obfuscation (englisch für *Software-Verschleierung*) ist seit vielen Jahren ein wichtiges Forschungsfeld im Bereich der IT-Sicherheit und beinhaltet etliche ungelöste Fragestellungen. Während theoretische Analysen demonstrieren, dass beweisbar sichere Software-Obfuscation im Allgemeinen nicht möglich ist, zeigen doch viele Einsatzgebiete in der Praxis (z.B. Schadsoftware, Schutzmechanismen für kommerzielle Software, usw.), dass Software-Obfuscation erfolgreich eingesetzt werden kann. Bis jetzt war jedoch größtenteils unerforscht, in welchem Ausmaß heutige Software-Obfuscation-Techniken mit dem technischen Fortschritt der letzten Jahre im Bereich der Programmcode-Analyse mithalten können. Der erste Teil dieser Dissertation untersucht aus diesem Grund, wie effektiv verschiedene Klassen von Software-Obfuscation gegenüber immer ausgefeilteren Deobfuscation-Techniken und kommerzieller Code-Analyse-Software sind. Dazu stellen wir ein neuartiges Klassifikationsschema für die Widerstandsfähigkeit von verschiedenen Arten von Software-Obfuscation in spezifischen Angriffsszenarien vor. Die Antwort auf die Fragestellung hängt stark von den Zielen des Angreifers und von den ihm zur Verfügung stehenden Ressourcen ab. Sogar einfachste Software-Obfuscation-Techniken können sehr wirkungsvoll gegenüber Analysetechniken sein, die auf Mustervergleich oder statischer Analyse basieren. Dies erklärt die ungebrochene Popularität von Software-Obfuscation bei Autoren von Schadsoftware. Dynamische Analysetechniken, insbesondere wenn diese durch einen menschlichen Analysten unterstützt werden, sind jedoch weitaus schwieriger beherrschbar. Dies macht den Einsatz von Software-Obfuscation zum Schutz geistigen Eigentums herausfordernd.

Im zweiten Teil dieser Dissertation beschäftigen wir uns aus diesem Grund mit dem Schutz von geistigem Eigentum durch Software-Obfuscation. Software-Diversifikation ist eine effektive Methode, mit der verhindert werden kann, dass eine automatische Attacke auf eine Programminstanz auch gegen andere Instanzen angewendet werden kann (engl. *class break*). Jedoch ist die Distribution von diversifizierter Software schwierig, da jede Kopie unterschiedlich sein muss. Das zweite Forschungsproblem, das wir in dieser Arbeit betrachten, ist die Entwicklung einer Methode zur Software-Diversifikation, die keine unterschiedlichen Instanzen benötigt. Wir stellen eine neuartige Software-Obfuscation-Technik vor, die das Konzept der Diversifikation auf den Kontrollflussgraphen eines Programms anwendet. Dieser Ansatz macht dynamische Angriffe deutlich schwieriger, da die Information, die von der Ausführung des Programms mit einem bestimmten Eingabewert erlangt werden kann, nicht zum Verständnis der Ausführung mit einem anderen Eingabewert beiträgt.

Während die tatsächliche Effektivität von Software-Obfuscation ausschließlich von den verfügbaren Ressourcen und der Geduld des Angreifers abhängt, bieten hardwarebasierte Technolo-

gien (*trusted computing*) deutlich mehr Möglichkeiten zum Schutz von Software. Jedoch wurde bis jetzt keine systematische Untersuchung des Zusammenspiels von hardware- und software-basierten Schutzmechanismen durchgeführt. Das dritte Forschungsproblem, das wir in dieser Arbeit zu lösen versuchen, ist die Unterstützung von Software-Obfuscation durch geringen Einsatz von Hardware. Wir stellen minimale Modifikationen von Intels AES-NI Instruktionsset vor, welche dessen Einsatz im Kontext von Software-Schutz möglich machen und verwenden diese neuen Instruktionen zur Parametrisierung unserer Kontrollflussgraph-Obfuscation-Technik. Das kombinierte Konzept ermöglicht die Bindung von Software an eine bestimmte Hardwareinstanz und schränkt die Möglichkeiten eines Angreifers auf eine dynamische Analyse des kompletten Programms ein — zwei wichtige Konzepte, um Programmanalyse effektiv zu erschweren.

Im letzten Teil dieser Dissertation fokussieren wir uns auf das Problem von Schadsoftware-Obfuscation. Vor einigen Jahren wurde des Konzept der semantischen Erkennung von Schadsoftware vorgestellt. Im Gegensatz zu syntaktischer Analyse (z.B. Vergleich eines Programms mit zuvor erstellen Signaturen von Schadsoftware-Proben) versucht die semantische Erkennung von Schadsoftware deren Effekte auf die ausführende Hardware zu modellieren und ist somit nicht auf eine bestimmte syntaktische Implementierung angewiesen. Dazu wird ein Modell der ausführenden Maschine verwendet. Das vierte Forschungsproblem, das wir in dieser Arbeit behandeln, ist die Implementierung von versteckter Funktionalität basierend auf Eigenschaften, deren Abbildung in einem Hardware-Modell schwierig ist. Wir stellen COVERT COMPUTATION vor, ein Konzept zur Implementierung von Funktionalität in Seiteneffekten des Mikroprozessors. Weiters präsentieren wir eine ausführliche Analyse von Seiteneffekten in der x86-Architektur und demonstrieren die Effektivität von COVERT COMPUTATION für Schadsoftware-Obfuscation.

Contents

CHAPTER 1

Introduction

Today, software is usually distributed in binary form which is, from an attacker's perspective, substantially harder to understand than source code. However, various techniques can be employed for analyzing binary code. For malware detection, simple pattern matching is applied to identify malicious functionality. In reverse-engineering scenarios, static as well as dynamic code analysis concepts, optionally assisted by a human analyst, are used to restore a higher-level representation (e.g. assembly code) of software in order to get a deeper understanding of its structure and behavior.

The term *software obfuscation* (or *code obfuscation*) covers a broad spectrum of techniques to obscure the control flow of software as well as data structures that contain sensitive information. Usually, software obfuscation is used to mitigate the threat of reverse-engineering. Collberg et al. [49] define an obfuscating transformation as any transformation of a source program into a target program, where both programs have the same observable behavior. The aim of an obfuscating transformation is to make the analysis of the target program more difficult. Another formal concept of software obfuscation by Barak et al. [12] describes an obfuscated program as a "virtual black-box", which ensures that no information can be derived from its analysis that cannot readily be computed by just observing its input/output behavior. Although this work shows that a universal obfuscator for any type of software does not exist and perfectly secure obfuscation is not possible in general, software obfuscation is still widely used to "raise the bar" for program analysis.

1.1 Motivation

The development of new code obfuscation schemes is always driven by the desire to hide the specific implementation of a program from increasingly sophisticated automatic or human-assisted analysis techniques of executable code. In early years, malware was the driving force in the steady refinement of code obfuscation schemes. The Brain computer virus, which was believed to be the first computer virus for MS DOS and was discovered in 1986 [7], can also be considered the first piece of software that implemented a technique for obfuscating the functionality

of its code. While the employed obfuscation was simple and just intercepted reads of the virus body to return innocuous code, it marked the beginning of an arms race between attackers and defenders. On the defender side, more and more sophisticated obfuscation techniques that hide behavior of code have been developed, while on the attacker side increasingly complex code analysis techniques allow to defeat obfuscations.

The underground economy's demand for stealthy malware is still one of the major driving forces behind the development of obfuscation techniques. The other leading application area of code obfuscation is the protection of commercial software against reverse-engineering. The motivations of an attacker can be diverse. An attacker might be interested in extracting some secret information from the program code that should not be revealed to the user. This secret might be a cryptographic key, a sophisticated algorithm that is considered a trade secret, or credentials for a remote service. Typical examples include DRM-enhanced media player software that store secret player keys required to decrypt content or incorporate purportedly secret algorithms (such as the cipher CSS used in the DVD copy protection framework). Another motivation for reverse-engineering is the modification of software to change its behavior. An attacker might want to use hidden program functionality, uncover firmware features that are disabled in low-cost devices or interface commercial software with own products; all these actions may interfere with the business model of the software vendor and may therefore be unwanted.

On the opponents side, research in the area of code analysis made tremendous progress during past years. Disassemblers, which extract the executable assembly code from binaries, became increasingly sophisticated, implementing complex heuristics or expensive static analysis methods to reconstruct code from a potentially obfuscated binary as precisely as possible. IDA Pro has become the industry standard for disassembly, used by reverse-engineers and malware analysts for manual code analysis. Research tools such as JAKSTAB [113] or BAP [24] allow to directly analyze binary code, to precisely reconstruct control flow, and reason about the behavior of the code. Furthermore, systems for dynamic malware analysis, such as ANUBIS [14], make a detailed analysis of the runtime behavior of a piece of code possible.

The first aim of this thesis is to analyze how effective different classes of obfuscations are in the presence of ever more sophisticated de-obfuscation techniques and off-the-shelf code analysis tools. Based on the analysis of the status quo and a novel classification method for code obfuscation schemes, we then explore new ways of protecting software. Thereby, we set our focus on hardware assistance. We aim at improving obfuscation techniques by using them in conjunction with characteristics of the underlying hardware. Thus, we strive to use hardware-based mechanisms to make reverse-engineering of software (i.e., finding an embedded private key) more difficult. Furthermore, this thesis analyzes ways to automatically diversify software on the control flow graph so that one copy of the software runs only on one host. The latter concept is a very promising approach for software protection, since a crack developed for one instance of a program will most likely not run on another instance and each individual copy of the software needs to be attacked independently. Finally, this thesis discusses the strength of state-of-the-art semantic-aware malware detection approaches in the presence of a novel malware obfuscation scheme which is based on side-effects of the underlying hardware.

1.2 Problem Statement

This thesis explores novel ways to improve the resilience of code obfuscation against different types of attacks. Hereby, we discuss four fundamental research questions in depth. First, it is difficult to formalize the resilience of code obfuscation. Thus, it remains unclear which type of code obfuscation provides reasonable protection against different types of attacks. Second, with the mass distribution of software class breaks become a more and more crucial problem. Third, previous software protection solutions can roughly be divided into two contrary categories: schemes using code obfuscation techniques and schemes that employ trusted computing technologies. The space between them has not gained much attention by the research community. Fourth, the suitability of side-effects of the hardware for malware obfuscation purposes has not been researched in the past. In the following, we discuss the four research questions in detail.

1.2.1 Evaluation of the Status Quo in the Arms Race between Software Obfuscation and Code Analysis

Since the groundbreaking work of Collberg et al. [49] on a taxonomy of obfuscation techniques, a vast number of new schemes have been proposed in the literature. Nevertheless, their security and effectiveness remain a controversially discussed topic. Theoretical results indeed indicate that "provably secure" obfuscation is generally impossible to achieve. Barak et al. [12] showed that it is impossible to construct a universal obfuscator that is applicable to any program; positive results are only known for very simple classes of functionalities such as point functions [29, 133, 205]. Beyond obfuscation of point function, other works on theoretical aspects of code obfuscation exist such as [59], [89], and [85]. These results, however, do not have much significance for the effectiveness of code obfuscation in practical scenarios.

Indeed, theoretical results are in stark contrast to the observation that code obfuscation is widely employed in practice: for example, almost all newly discovered malware comes with some form of obfuscation to hide its functionality, and commercial software products (such as Skype or DRM engines) still employ obfuscation techniques as part of their protection portfolio. Consequently, a plethora of obfuscation techniques have been described in the literature (e.g. [49, 71, 107, 201]); most of them try to "raise the bar" for reverse-engineering attempts, but do not come with a rigorous security analysis, let alone a proof. Some attempts have been made to quantify the additional complexity that is added to an executable by employing code obfuscation techniques (for example, see [135]); however it remains unclear whether such notions indeed capture all security properties of obfuscation correctly. While this is clearly unsatisfactory from a theoretical point of view, it is probably the best one can achieve with current knowledge.

Thus, it remains largely unexplored to what extent code obfuscation techniques developed in the last two decades can keep up with the progress in code analysis. The first research problem of this thesis investigates where we stand in the arms race between attackers and defenders and is further concerned with methods for the evaluation of different types of obfuscations in specific attack scenarios.

1.2.2 Control Flow Graph Diversification

Software diversity is a concept that aims at delivering syntactically different but semantically identical versions of a program to different users in order to prevent so-called class breaks. Automated attacks developed against one instance of a program are thus unlikely to work against a differently obfuscated version [3, 4]. While software diversification can be considered as a very effective solution against class breaks, it raises major difficulties in software distribution, as individual instances of the program have to be unique. Thus, no efficient way for the distribution of diversified copies via physical media (e.g. DVD) exists.

The second research problem of this thesis is the development of a novel concept for control flow diversification which do not require the individual copies of the program to be unique on the binary code layer and thus solve the distribution problem.

1.2.3 Using Hardware Properties for Code Obfuscation

While code obfuscation has been under heavy research for many years, up to now no obfuscation technique has been able to provide a well-defined level of security and theoretical results indicate that provably secure obfuscation in general is not possible [12]. Thus, the resilience of software-only code obfuscation remains unclear and ultimately depends only on available resources and patience of the attacker. Hardware-based solutions (*trusted computing*) provide a much wider range of protection mechanisms such as remote attestation and secure storage for secrets; however, a significant amount of additional hardware is required.

Almost no systematic research has been done on hardware-assisted code obfuscation. Thus, the third research problem is concerned with methods for supporting code obfuscation with small protection mechanisms in hardware.

1.2.4 Obfuscation against Semantic-aware Malware Detection

Malware obfuscation and analysis have become important areas in academic research. For client-based malware detection (commonly known as virus scanners), the analysis methods have not changed fundamentally during the last years. Static code analysis is still the predominant approach for the classification of the maliciousness of programs and follows the simple approach of signature matching [42, 91]. Thereby, the analysis of the code is primary performed on a syntactical layer. Thus, only the occurrence of certain instructions or sequences of instructions is taken into account for the evaluation of maliciousness. However, the semantics of these instructions are ignored so that the meaning of the code remains unknown to the detection system and thus is out of scope of its basis for classification decision-making. In addition to malware signatures, most of today's commercial virus scanners employ basic heuristic detection approaches, which search for more generic patterns of potentially malicious behavior and are located at a more semantic level. Thus, these systems are more resilient to simple code obfuscation. Still, their effectiveness has to be considered as rather low [187]. The idea of semantic-aware malware detection [43] goes a step further by making the pattern (which is called template in this approach) for maliciousness independent from its actual implementation. For example, a template can describe the functionality of adding a value to a register without defining which particular register

is being used in the actual implementation. This concept makes the matching much more robust against code obfuscation techniques that modify the program's syntactical representation (e.g., changing the layout of the code).

The problem with this approach, however, is that the underlying model of the real world (i.e. the microprocessor) is incomplete (simply because of the fact that a model is an abstract, simplified representation of a real object). In an incomplete model, not all effects a program's sequence of instructions might have on the real hardware, can be identified, thus leaving possibility of implementing functionality that is not directly visible in the model. The fourth research problem develops methods for hiding potentially malicious code in harmless looking code using side effects of the underlying hardware that are not covered by the analysis model.

1.3 Main Results

The main results of this thesis are in line with the previously defined research problems. The first main result is an evaluation of how effective today's state-of-the-art code obfuscation techniques are in the presence of ever more sophisticated de-obfuscation techniques and off-the-shelf code analysis tools using a novel classification scheme that combines the attacker's aims with available code analysis techniques (Chapter 2). The second main result consists of the development of a parameterizable control flow graph obfuscation scheme that makes software diversification possible without requiring each individual copy to contain different code (Chapter 3). The third main result is the development of a concept for hardware-assisted code obfuscation (Chapter 4). It is based on modifications to Intel's AES-NI instruction set extension which make them usable in white-box attack scenarios and incorporates our control flow obfuscation techniques introduced in Chapter 3. The fourth main result is an in-depth analysis of side-effects in the system's microprocessor for obfuscation purposes in a malware context (Chapter 5). In the following, the four main results are presented in detail.

1.3.1 Definition of Attack Scenarios and Classification of Obfuscation Techniques

We introduce a novel classification scheme for code obfuscations to analyze the fundamental question of which classes of code obfuscation can provide reasonable protection against today's state-of-the-art code analysis techniques and tools. The classification scheme is based on specific attack scenarios which are a combination of the attacker's aims and available code analysis techniques. Even very simple obfuscation techniques are quite effective against pattern matching or static analysis concepts which are typically used in malware analysis scenarios. In contrast, dynamic analysis methods, in particular if assisted by a human analyst, are considerably stronger against many classes of obfuscations. This makes code obfuscation for the protection of intellectual property highly challenging.

In summary, our main contributions in this part of the thesis are:

- We introduce a novel classification scheme for code obfuscations which is based on specific attack scenarios.

- We systematically analyze literature on software obfuscation as well as code analysis and evaluate which classes of code obfuscation can provide reasonable protection in particular attack scenarios.

1.3.2 Development of a Control Flow Graph Diversification Scheme

We propose a novel code obfuscation scheme which provides strong protection against automated static reverse-engineering and uses the concept of software diversification in order to enhance the complexity of dynamic analysis. Diversification is applied to the control flow graph of the software and does not require individual copies of the program to be unique on the binary code layer. By splitting code into small blocks before diversification, we achieve a complex control flow graph and static analysis can only reveal very limited local information of the program. We further provide a practical evaluation of our concept that demonstrates its strength against automated deobfuscators and we show that it can dramatically increase the effort for an attacker, because knowledge derived from one run of the software does not necessarily help in understanding the behavior of the software in runs on other inputs.

In summary, our main contributions in this part of the thesis are:

- We introduce the novel concept of control flow graph diversification for software protection.
- Based on a prototype implementation we demonstrate how our approach makes program analysis considerably more time-consuming while causing reasonable performance overhead.

1.3.3 AES-SEC: Modification of AES-NI Instructions for Application in a White-box Analysis Context

To tackle the dilemma of impossibility results for code obfuscation, we propose the idea of assisting an obfuscation technique with small protection mechanisms in the system's microprocessor. Unlike heavy, full-fledged hardware such as dedicated co-processors our concept requires only moderate modifications to the hardware and could be easily implemented in future generations of microprocessors. In detail, we propose modifications to Intel's AES-NI instruction set that make them suitable in software protection scenarios. Further, we demonstrate how these modified instructions can be used in conjunction with the control flow graph obfuscation scheme introduced in Chapter 3. Finally, we propose a key distribution system for our hardware-assisted code obfuscation scheme.

In summary, our main contributions in this part of the thesis are:

- We introduce AES-SEC, a modification of the AES-NI instruction set extension of today's Intel processor architectures.
- We propose a key distribution scheme based on AES-SEC which allows to securely place cryptographic keys in the customer's hardware.

- We combine AES-SEC with our control flow obfuscation approach (Chapter 3) into a novel hardware-assisted obfuscation scheme.

1.3.4 Covert Computation – Hiding Code in Code

In contrast to signature matching, semantic-aware malware detection [43] is more resistant against simple classes of obfuscating transformations such as *garbage insertion* [49] and *equivalent instruction replacement* [69]. 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 thesis we show that exactly this fundamental prerequisite for semantic-aware malware detection is difficult to achieve. We introduce a novel 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 thesis 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 summary, our main contributions in this area are:

- We introduce a novel approach for code obfuscation called COVERT COMPUTATION which is based on side effects in today's microprocessor architectures.
- 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.

1.4 Related Work

In this section, we discuss related work in the research areas of this thesis. In addition to this section, Chapter 2 provides an in-depth literature survey which compares today's software obfuscation with the state-of-the-art in code analysis.

1.4.1 Software Protection through Code Obfuscation

Throughout the last two decades there have been a large number of publications the protection of intellectual property in software. These solutions can roughly be divided into the two categories

code obfuscation and trusted computing technologies. For the first category, a comprehensive taxonomy of obfuscating transformations was introduced in 1997 by Collberg et al. [49]. To measure the effect of an obfuscating transformation, Collberg et al. defined three metrics: *potency*, *resilience* and *cost*. *Potency* describes how much more difficult the obfuscated program P' is to understand for humans. Software complexity metrics (e.g. [36, 96, 98, 99, 142, 151, 157]), which were developed to reduce the complexity of software, can be used to evaluate this rather subjective metric. In contrast to potency that evaluates the strength of the obfuscating transformation against humans, *resilience* defines how well it withstands an attack of an automatic deobfuscator. This metric evaluates both the programmer effort (how much effort is required to develop a deobfuscator) and the deobfuscator effort (the effort of space and time required for the deobfuscator to run). A perfect obfuscating transformation has high potency and resilience values, but low *costs* in terms of additional memory usage and increased execution time. In practice, a trade-off between resilience/potency and costs (computational overhead) has to be made. However, the main problem of measuring an obfuscation technique's strength is that a well-defined level of security does not exist, even though it can make the process of reverse-engineering significantly harder and more time consuming. Moreover, in 2001 Barak et al. [12] proved that there exist classes of functions (programs) that cannot be obfuscated by any type of obfuscator. Although some of the conclusions are highly controversial, the work shows that a universal obfuscator for any type of software does not exist. Several other theoretical works on software obfuscation appeared, such as [133] and [205], but practical relevance is limited.

As preventing disassembling is nearly impossible in scenarios where attackers have full control over the host on which the software is running, the common solution is to make the result of disassembling worthless for further static analysis by preventing the reconstruction of the control flow graph. To this end, Linn and Debray [130] and Cappaert and Preneel [32] use so-called branching functions to obfuscate the targets of `CALL` instructions: The described methods replace `CALL` instructions with jumps (`JMP`) to a generic function (*branching function*), which decides at runtime which function to call. Under the assumption that for a static analyzer the branching function is a black box, the call target is not revealed until the actual execution of the code. This effectively prevents reconstruction of the control flow graph using static analysis. However, the concept of a branching function does not protect against dynamic analysis. An attacker can still run the software on various inputs and observe its behavior. Madou et al. [134] argue that recently proposed software protection models would not withstand attacks that combine static and dynamic analysis techniques. Still, code obfuscation can make dynamic analysis considerably harder.

In recent literature, code obfuscation is experiencing a comeback in the field of attack prevention [67, 81]. An attack is called a class break, if it was developed for a single entity, but can easily be extended to break any similar entity. In software, for example, we would speak of a class break if an attacker can not only remove a copy protection mechanism on the software purchased, but also can write a generic patch that removes it from every copy of the software. For software publishers, class breaks are dreaded, because they allow mass distribution of software cracks (e.g. on the Internet) to people who would otherwise not be able to develop cracks themselves. The concept of diversification for preventing class breaks of software was put forth by Anckaert [2]. An algorithm for automated software diversification was introduced

by De Sutter et al. [69]. Their approach uses optimization techniques to generate different, but semantically equivalent, assembly instructions from code sequences. While software diversification is an effective solution for software protection (see e.g. [3]), there is no practical way for distributing diversified copies via physical media (e.g. DVD), and software updates for diversified software are difficult to distribute as well. Franz [81] proposes a model for the distribution of diversified software on a large scale. The author argues that the increasing popularity of online software delivery makes it feasible to send each user a different version of the software. However, a specific algorithm for the diversification process is not given.

Another approach to protect cryptographic keys embedded in software is the use of white-box cryptography which attempts to construct a decryption routine that is resistant against a "white-box" attacker, who is able to observe every step of the decryption process. In white-box cryptography, the cipher is implemented as a randomized network of key dependent lookup tables. A white-box DES implementation was introduced by Chow et al. [40]. Based on this approach, other white-box implementations of DES and AES have been proposed, but all of them have been broken so far (see e.g. Jacob et al. [105], Wyseur et al. [213] and Billet et al. [17]). Michiels and Gorissen [145] introduce a technique that uses white-box cryptography to make software tamper-resistant. In their approach, the executable code of the software is used in a white-box lookup table for the cryptographic key. Changing the code would result in an invalid key. However, due to the lack of secure white-box implementations, the security of this construction is unclear.

1.4.2 Hardware-assisted Code Obfuscation

With attestation features and secure storage of secrets (e.g., cryptographic keys) in software, trusted computing provides significantly more technical capabilities for software protection compared to software-only protection mechanisms. Röder et al. [164] described HADES, a client-server environment, which uses trusted computing to assure the integrity of a system before sensitive keys are exchanged. However, they do not consider protection of keys against attackers that compromise the system after the initial measurement process has been performed. Cooper and Martin [55] described another general architecture of a trusted computing based platform, without providing an actual implementation. FLICKER [143] is a virtualization technique for secure code execution on a legacy operating system. Similarly, TRUSTVISOR [144] is based on a hypervisor that provides code and data integrity as well as secrecy for sensitive parts of a program. CARMA [198] is a secure execution environment based on commodity x86 hardware. SICE [8] makes use of multicore CPUs to allow isolated program execution. In 2004, ARM introduced the security extension TRUSTZONE [1] which allows to run security-relevant code in a separate area of the processor (*secure world)*, whereas other code runs in a less trusted area (*normal world*).

A very simple hardware-based solution against software piracy is the use of hardware tokens (dongles). Typically, the software checks for the existence of a dongle from time to time. However, token-based solutions can be easily circumvented by removing these checks from the software. Two types of attacks on dongles were introduced by Mitchell [147]. The strength of dongles for software protection was evaluated by Piazzalunga et al. [158]. The authors con-

cluded that hardware tokens provide only very limited protection. In Section 2.4 dongle-based protection mechanisms are evaluated in the context of specific attack scenarios.

Little systematic research has been done on hardware-supported code obfuscation. A combination of hardware and software obfuscation was introduced by Fu et al. [82]. In their approach, parts of the security relevant code are protected by traditional software-only obfuscation which significantly reduces the strength of the entire system. Zhuang et al. [217] address attacks on software, caused by the exposure of the control flow in system memory. By implementing a randomizer in the processor, software blocks (e.g. loops) are written to random locations, thus obfuscating the control flow. Fukushima et al. [83] proposed a hardware-assisted obfuscation scheme for mobile phones. An applet that runs on the SIM card provides secret parameters for the correct execution of the obfuscated program. Chakraborty et al. [34] introduced a control flow obfuscation scheme that is based on distributed verification of an input-dependent set of keys. Further, the authors demonstrate how this approach can be combined with trusted computing technologies. In 2013 Gueron [94] proposed a white-box implementation of AES which uses Intel's AES-NI instruction set. It is aimed at protecting the cryptographic key even in a white-box analysis context, where the attacker has full control over the system. However, as the authors state, the approach requires additional software-only code obfuscation. Thus, the resilience of the concept as a whole remains unclear.

In summary, while code obfuscation has unclear security properties, trusted computing concepts are backed by a more solid theoretical foundation. However, significant amount of additional hardware is required.

1.4.3 Malware Obfuscation and Analysis

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 [153]. At runtime, an unpacking routine is used to transform the data block back into machine-interpretable code. Polymorphism [184] and metamorphism [156] 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. [210]. 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 [132]. Bruschi et al. [25] 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. [114]. Christodorescu et al. [43] 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 [61] and 2008 [62]. An important aspect of this type of malware detection is that it 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 [85]. However, no practical approach of this idea was given in the paper. Moser et al. [149] 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 scheme 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 [208]. Sharif et al. [177] use Windows API call monitoring for deobfuscation. Malware detection using symbolic execution was put forth by Crandall et al. [57]. Bayer et al. [15] 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 [15], which were extracted by running malware in a controlled environment [117]. Furthermore, dynamic malware analysis [74] has become an important concept to assist 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 [122], this method still requires reverse-engineering of obfuscated binaries in order to uncover crucial routines such as domain generation algorithms [203], obfuscated encryption methods, or cryptographic keys [40].

1.5 Structure of the Work

In Chapter 2 we survey the state-of-the-art in software obfuscation as well as code analysis. Giving answers to the question of which software obfuscation techniques can be applied in which particular scenario is the first main contribution of this thesis. Section 2.1 defines *attack scenarios* which are based on the combination of the attacker's aim for analyzing a program and the code analysis methods and tools that are available to him. In Section 2.2 we give a comprehensive overview of software obfuscation techniques that can be found in the literature from the last two decades. Section 2.3 describes the state-of-the-art in static, dynamic and human-assisted code analysis both from an academic and an industry perspective. Section 2.4 is

the main part of Chapter 2. It provides a comprehensive analysis of the robustness of different classes of obfuscations against available code analysis and de-obfuscation techniques in the context of particular attack scenarios. The results of this chapter make it possible to determine if a class of code obfuscations is suitable for the protection of programs in different attack scenarios.

In Chapter 3 we introduce a novel code obfuscation scheme which is based on diversification of the program's control flow. Our concept solves the problem of distribution of diversified instances of a program, as diversification is applied to the control flow graph only. Thus, each instance of the program looks the same. Section 3.1 explains the concepts required for control flow diversification. In Section 3.2 we discuss performance and security implications of our approach.

In Chapter 4 we propose modifications to the AES-NI instruction set in Intel microprocessors and combine them with our code obfuscation approach from Chapter 3. Section 4.2 explains the scenario in which our obfuscation scheme can be used for software protection. In Section 4.3 we describe AES-SEC, a modified version of the AES-NI instruction set extension. AES-SEC allows the implementation of cryptographic calculations in a white-box analysis context where an attacker has full control over the system the protected program is running on. The application of AES-SEC for control flow obfuscation is described in Section 4.4 and a performance evaluation is given in Section 4.5.

In Chapter 5 we introduce our concept of COVERT COMPUTATION. We show how side effects in the microprocessor can be used to hide code in code and discuss resulting challenges for malware detection. Section 5.1 provides a comprehensive analysis of side effects in Intel's x86 architecture and demonstrates how these side effects can be used to implement hidden functionality. In Section 5.2 we introduce a prototype implementation of our COVERT COMPUTATION concept that is based on compile-time obfuscation in LLVM. In Section 5.3 a rigorous security analysis of our concept is given where we analyze its resilience against semantic-aware malware detection. Finally, Section 5.4 evaluates performance implications of COVERT COMPUTATION.

1.5.1 Publications

This thesis is based on the following publications:

- Sebastian Schrittwieser and Stefan Katzenbeisser. *Code Obfuscation against Static and Dynamic Reverse Engineering*. In Proceedings of the 13th International Conference on Information Hiding, pages 270–284. Springer, 2011. [171]

- Sebastian Schrittwieser, Stefan Katzenbeisser, Peter Kieseberg, Markus Huber, Manuel Leithner, Martin Mulazzani, and Edgar Weippl. *Covert Computation: Hiding Code in Code for Obfuscation Purposes*. In Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pages 529–534. ACM, 2013. [172]

- Sebastian Schrittwieser, Stefan Katzenbeisser, Peter Kieseberg, Markus Huber, Manuel Leithner, Martin Mulazzani, and Edgar Weippl. *Covert Computation – Hiding Code in Code through Compile-Time Obfuscation*. Computers & Security, 42(0):13 – 26, 2014. ISSN 0167-4048. [173]

- Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, and Edgar Weippl. *25 Years of Software Obfuscation. Can it keep pace with Progress in Code Analysis?*. Under submission.
- Sebastian Schrittwieser, Stefan Katzenbeisser, Georg Merzdovnik, Peter Kieseberg, and Edgar Weippl. *AES-SEC: Improving software obfuscation through hardware-assistance*. Under submission.

CHAPTER 2

Literature Review and Classification of Obfuscation Techniques

Software obfuscation has always been a highly controversially discussed research area. Still, it is not clear how strong today's software obfuscation state-of-the-art is in the presence of more and more sophisticated code analysis concepts.

To this end, we first analyze and classify different attack scenarios; in particular, we distinguish between methods that an attacker is willing to employ during his attack as well as his aims. A combination of both, termed *scenarios* in the sequel, forms a basic attacker model. Subsequently, we describe and categorize existing code obfuscation techniques; we limit ourselves to obfuscations that are applicable to binary code or byte code, as they are the most prominently used ones. We therefore decided to leave obfuscation techniques targeting source code (such as removal of comments or renaming of variables to make a program "unreadable"), out of scope, as they are rarely used and have questionable effectiveness and security. We then discuss the state-of-the-art in program analysis methods and particularly focuses on their capabilities and limits. In Section 2.4, which forms the core of this chapter, we evaluate the security of code obfuscation techniques against the different scenarios, taking into account recently published attacks as well as recent off-the-shelf program analysis tools.

2.1 Attack Scenarios

The most general view of program obfuscation, as put forward by Barak et al. [12], is to treat an obfuscated program as "virtual black-box", which reacts with a result once it is invoked on some input values. An obfuscation can be seen secure in this model if no information can be derived from the knowledge of the obfuscated program that cannot readily be computed by just analyzing its input/output behavior. While being of high theoretic interest (the definition allowed to prove a general impossibility result), it is unsuitable for real-life settings. In practice, actual applications demand properties which are less strong than the virtual black-box paradigm: for

example, a defender may not care if an attacker derives *some* knowledge on the program, except for a most valuable item such as a cryptographic key or a secret subroutine.

In this section we present a novel classification of real-life attack scenarios in the context of code obfuscation, derived from a careful analysis of past security incidents involving obfuscated programs. We first distinguish between various analysis techniques that an attacker is willing to employ during his attack; later we deal with different aims of an attacker. Combining these two concepts, we arrive at attack scenarios, which will be used to analyze obfuscations in subsequent sections of this chapter.

2.1.1 Code analysis categories

We categorize code analysis techniques in four general classes of increasing sophistication and complexity. Depending on the aim of an attacker, available resources, and time constraints different analysis techniques can be used. For example, a human reverse-engineer who tries to understand a piece of code of a competitor may afford spending time and effort on highly complex and time-consuming analyses, while an anti-virus vendor, who has to timely analyze hundreds of thousands of different malware samples each day, may be required to resort to very lightweight and thus limited analysis techniques.

Pattern Matching Pattern matching is the most simple form of code analysis, where one looks for the existence of a particular sequence of instructions in a program's code. Today, anti-virus products are mostly based on such fast pattern matching techniques. The state-of-the-art in pattern matching is difficult to determine as most algorithms are considered trade secrets of anti-malware vendors. However, it is safe to assume that currently employed techniques are stronger than simple matching of static patterns of instructions, but weaker than regular expressions. For example, Christodorescu et al. [43] described the concept of semantics-aware malware detection, which utilizes so-called templates of malicious code. The technique can be considered as one of today's more sophisticated pattern matching approaches; in the sequel, we thus use this approach as prototypical instantiation of code analysis by pattern matching.

Automated static analysis Static analysis works by analyzing the code of a program without actually executing it. The primary goal is to reconstruct some higher-level semantic structures of the program, such as its control flow graph or its data sections (see Section 2.3). To some extent, static analysis techniques are implemented by today's anti-virus scanners under the key term *static heuristic malware detection*.

Automated dynamic analysis In contrast to static analysis, dynamic analysis runs a program on a particular set of input values, observes its actions and collects all available information such as executed instructions, issued system calls or accessed memory locations (see Section 2.3). Dynamic analysis allows a deeper understanding of the program's behavior. However, the data gathered from running the program on one or more inputs does not necessarily allow to draw conclusions about the behavior of the entire program. This is particularly true for malware that is

only activated once it receives an external stimulus. A typical application area for automated dynamic analysis can be found in the labs of anti-virus vendors, where signatures for new malware samples are dynamically generated in an almost entirely automated process. In recent years, a multitude of code analysis approaches that incorporate both static and dynamic techniques have emerged (e.g., McVeto by Madou et al. [134], Thakur et al. [189]). We include approaches following this mixed type in the class of *automated dynamic analysis*.

Human-assisted analysis On the top end of the scale we find a human analyst that performs a tool-assisted exploration of a piece of code; the utilized tools can either be based on static or dynamic analysis. Typically, this attack is referred to as reverse-engineering, where the attacker aims at getting full understanding of a program's structure and behavior utilizing tools such as IDA Pro [73].

2.1.2 Attacker's aims

This section systematically categorizes and characterizes motivations of an attacker for analyzing software and sorts them according to increasing complexity. We believe that almost all attacker goals observed in practice fit in one of these four general categories. As a running example to illustrate the goals throughout the section we use a program implementing a cryptographic algorithm with an embedded secret key. We further present other examples for each category in order to demonstrate practicability of this classification.

Finding the location of data The analyzer aims at retrieving some data embedded in the program in its original, non-obfuscated representation from the obfuscated program. In our running example the analyzer may want to extract the secret cryptographic key from the obfuscated program in order to be able to decrypt data in a different context than provided by the application (e.g. to circumvent DRM policies). Other typical examples that fall into this category are the extraction of licensing keys, certificates, credentials for remote services and device configuration data.

Finding the location of program functionality The analyzer aims at identifying the entry point of a particular function within an obfuscated program. In our running example, the analyzer may want to find the entry point of the cryptographic algorithm contained in the obfuscated program in order to analyze it in subsequent steps. Another aim could be finding the exact location of a copy protection mechanism (such as a check for the presence of a hardware dongle or the validation of a licensing key) in order to circumvent it. Furthermore, finding the code representation of a particular functionality of a program can be useful for manual reverse-engineering attacks on small areas of the program. More generally, one may ask the question whether a program implements a particular functionality at all (such as the AES encryption algorithm) or simply whether a program is malicious or not.

Another related aim of the attacker might be to modify the behavior of a program in a particular way (e.g., bypassing a copy protection mechanism). However, we see this out of scope of this thesis, because it is only marginally related to code obfuscation itself but rather

falls into the field of tamper-proofing (e.g., [101] and [35]). Still, *finding the location of program functionality* is a fundamental prerequisite of this aim.

Extraction of code fragments The analyzer aims at extracting a piece of code including all possible dependencies that implements a particular functionality from the obfuscated program. In our running example, the analyzer's aim can be the extraction of the cryptographic algorithm in order to build his own decryption routine. Note that for this purpose it is not necessary to fully understand the code, just using it in a new application may be enough. This approach is widely used for breaking DRM implementations. Instead of understanding how precisely the decryption routine embedded into the player works, it is simply extracted and included in the attacker's player, which decrypts the digital media without enforcing the contained usage policies. Another aim of the attacker may lie in the extraction of fragments of commercial software of competitors.

Understanding the program The analyzer aims at fully understanding a non-trivial fragment or even the entire obfuscated program. This requires that the attacker must be able to remove the applied obfuscation techniques and gain full understanding of the original, non-obfuscated program. In our running example, the analyzer may want to understand how a proprietary cipher embedded into the obfuscated program works in order to start cryptanalysis attempts. Another motivation for trying to understand a program can be the desire of an attacker to create new programs that are compatible with propriety software. Finally, intellectual property theft (e.g. gaining an understanding of file formats, protocols, etc.) is a major driving force for human-assisted reverse-engineering attacks.

2.1.3 Scenarios

Based on the combination of a code analysis technique with one of the attacker's aims, we arrive at a number of attack scenarios which form the basis of our subsequent analysis. As not all combinations are reasonable (e.g. pattern matching provides information on the code but cannot be used for extracting code), a total of 14 scenarios must be considered. An overview of the literature that describes code analysis techniques in these scenarios is provided in Table 2.1. In the following, the scenarios are described in more detail.

Locating data through pattern matching Patterns for the automated identification of data inside a program's code describe the structure of the data such as its length, data type, etc. or its environment in the form of code surrounding it. In this scenario a pattern matching algorithm is used to determine the existence and location of data that conforms to the pattern's specifications.

Locating code through pattern matching The most simple form of pattern matching for code are fixed patterns of sequences of instructions. More complex patterns, describing the behavior of code, are possible as well and can be implemented through code normalization techniques [44]. While in theory, patterns for normalized code can match functionality independent

from its actual implementation in code, existing implementations of this concept are limited to simple obfuscation techniques only.

Locating data through static analysis Static analysis techniques for data are usually based on the reconstruction of code: the location of data is determined through code sections that are accessing this data. For example, the observation of parameters for function calls can reveal a cryptographic key that is used to decrypt DRM-protected media, a key that is sent to an external device, or a token that is transmitted over a network.

Locating code through static analysis Locating code through static analysis is done by reconstructing the control flow graph in order to get a better understanding of the program's structure. For example, the control flow characteristics of a cryptographic algorithm can reveal its location in a binary.

Extracting code through static analysis Static analysis provides a conservative view on a program. The extraction of a particular functionality as well as its dependencies results in an *over-approximation* of the set of control flow edges (see Section 2.3). Thus, it usually contains behavior that does not occur in real executions of the program.

Understanding code through static analysis This scenario targets static de-obfuscation techniques that are able to transform the obfuscated code into a representation from which a human analyst can understand the program's functionality with reasonable efforts.

Locating data through dynamic analysis Dynamic analysis of data used by a program at runtime can be done by observation of the program's memory, stack, registers, parameters of system calls, etc.

Locating code through dynamic analysis Dynamic analysis reveals a program's behavior at runtime as well as the interaction with its environment (e.g. system calls). In this scenario, a particular functionality can be located through its specific runtime behavior.

Extracting code through dynamic analysis In contrast to static analysis, dynamic analysis allows a non-conservative view on the program. Code extracted from a run on a particular input does not necessarily have to contain the code required for a run of the program on another input and the resulting control flow graph is a subset of all possible execution paths. Thus, when dynamically extracting code from a program, the resolution of dependencies is a challenging task.

Understanding code through dynamic analysis Analogous to the scenario of understanding code through static analysis, this scenario targets automated dynamic de-obfuscation techniques that are able to transform the obfuscated code into a representation from which a human analyst can understand the program's functionality with reasonable efforts.

Scenarios with human assisted analysis Static and/or dynamic approaches assisted by a human analyst aim at getting full understanding of a particular aspect of the program. This aspect can be data in its pure form (data de-obfuscation), the location of code implementing a particular functionality or its dependencies as well as the entire program itself.

2.2 Software Obfuscation

In this section we briefly describe various code obfuscation schemes that were reported in the literature and classify them (in increasing order of complexity) into the three categories *data obfuscation*, *static code rewriting*, and *dynamic code rewriting*. Many of the described obfuscation techniques are folklore. Consequently, it is hard to pay tribute to the original source; we did this wherever possible. More details on early techniques can be found in Collberg's taxonomy of code obfuscation [49] and [48].

2.2.1 Data obfuscation

Code obfuscation techniques of this category modify the form in which data is stored in a program in order to hide it from direct analysis. Usually, data obfuscation requires the program code to be modified as well in order to be able to reconstruct the original data representation at runtime. Many data obfuscation techniques were first described by Collberg et al. [50].

Reordering of data Variables can be split into two or more pieces in order to make it more difficult for an attacker to identify them. The mapping between an actual value of a variable and its split representation is managed by two functions. While the first one is executed at obfuscation time, the other one reconstructs the original value of a variable from its split parts at runtime. For example, boolean variables can be obfuscated by splitting them into multiple boolean values. At runtime the variable's actual value is retrieved by performing a specific boolean operation (such as a logical XOR) over the parts of the variable. Other data types such as integers and string variables can be obfuscated in a similar way. In contrast to variable splitting, variable merging combines two or more variables into one.

In order to obfuscate the structure of an array it can be *split* into two or more subarrays. Conversely, multiple arrays can be *merged* into one. *Folding* (increasing the number of dimensions of the array) and *flattening* (decreasing the number of dimensions) are similar techniques which can be used for obfuscating the structure of data stored in arrays.

Obfuscating the structure of data by reordering its components to decrease locality (logically related items are physically close in the binary) is another fundamental obfuscation technique. For example, this obfuscation is often applied to cryptographic keys stored within commercial software.

A low-level implementation of data reordering for obfuscation purposes was introduced by Anckaert et al. [6]. By redirecting memory access though a software-based dispatcher the order of data in memory can be shuffled on a periodic basis, thus making its identification and analysis more difficult.

	Pattern Matching	Automatic Static	Automatic Dynamic	Human Analysis
Locating Data	Moser et al. [149]	Shamir and Van Someren [176] Kinder and Veith [113] Kinder [112]	Cozzie et al. [56] Lin et al. [127] Slowinska et al. [180] Zhao et al. [216]	Jacob et al. [105] Link et al. [129] Billet et al. [17] Wyseur and Preneel [212] Goubin et al. [90] Piazzalunga et al. [158] Wyseur et al. [213] Michiels et al. [146] Saxena et al. [170] Wyseur [211] De Mulder et al. [68]
Locating Code	Newsome et al. [154] Moser et al. [149] Tang and Chen [188] Griffin et al. [91] Dalla Preda et al. [63]	Harris and Miller [97] Bruschi et al. [25] Chouchane and Lakhotia [38] Dalla Preda et al. [60] Walenstein et al. [200] Bilar [16] Karnik et al. [110] Coogan et al. [53] Treadwell and Zhou [194] Tsai et al. [195] Jacob et al. [104] Kinder [112]	Deprez and Lakhotia [72] Madou et al. [134] Royal et al. [168] Wilde and Scully [206] Moser et al. [148] Sharif et al. [177] Song et al. [183] Li et al. [126] Sharif et al. [178] Webster and Malcolm [204] Comparetti et al. [52] Debray and Patel [70] Yin and Song [214] Coogan et al. [54] Gröbert et al. [92] Calvet et al. [28] Zeng et al. [215]	Madou et al. [137] Madou et al. [138] Rolles [165] Quist and Liebrock [160]
Extracting Code	*invalid scenario*	Sneed [182]	Leder et al. [125] Sharif et al. [178] Caballero et al. [26] Kolbitsch et al. [118] Zeng et al. [215]	Ning et al. [155] Canfora et al. [30] Field et al. [79] Cimitile et al. [46] Lanubile and Visaggio [121] Canfora et al. [31] Danicic et al. [64] Fox et al. [80] Danicic et al. [65]
Understanding Code	*invalid scenario*	Rugaber et al. [169] Majumdar et al. [139] Raber and Laspe [161] Guillot and Gazet [95]	Udupa et al. [197]	Biondi and Desclaux [18] Eagle [73] Myska [152] Kholia and Wegrzyn [111]

Table 2.1: Literature on code analysis in the 14 scenarios.

Encoding Static data (such as strings) within binaries contains useful information for an attacker. Under this obfuscation technique data is converted to a different representation with some special encoding function. At runtime, the inverse function is used to decode the data. Without analyzing the decoding function the original representation of the data and therefore its value is hidden.

Converting static data to procedures This obfuscation method replaces static data with a program that calculates the data at runtime. For example, a string object can be built at runtime, so that an attacker is not able to extract its value by examining the binary.

An extreme form of this obfuscation method is white-box cryptography. Its basic idea is to merge a secret key with elements of the cipher (e.g. the S-boxes), so that the key cannot be found in the binary anymore. The first white-box implementations of DES [40] and AES [40] were proposed by Chow et al., followed by other approaches [22, 128, 212].

2.2.2 Static code rewriting

Static rewriters are similar to compilers as they modify a program's code during obfuscation while its output is executed without further run-time modifications. Strictly speaking, all data obfuscation techniques described above would also fall into the category of static code rewriting. However, as the obfuscation targets are very different (data vs. binary code), thus requiring distinct obfuscation techniques, we decided to use separate categories for data obfuscation and static code rewriting.

Replacing instructions Often, a specific behavior of a program can be implemented in multiple ways and instructions or sequences of instructions can be replaced with semantically equivalent code. For example, on the Intel x86 platform the instructions `MOV EAX, 0` and `XOR EAX, EAX` are equivalent and can be replaced with each other. De Sutter et al. [69] replaced infrequently used opcodes with blocks of more frequently used ones in order to reduce the total number of different opcodes used in the code and to normalize their frequency. In Section 5, we introduce the idea of hiding potentially malicious code in side effects of innocent looking sequences of instructions. A side effect can be any effect on the state of the underlying machine that is not covered by the analysis model (e.g., the state of the flags register).

Opaque predicates A predicate (boolean-valued function) is opaque if its outcome is known to the obfuscator at obfuscation time, but difficult to determine for a de-obfuscator [49, 51]. Opaque predicates are used to make static reverse engineering more complex by introducing an analysis problem which is difficult to solve without running the program. The prime example for the use of opaque predicates is the obfuscation of a program's control flow graph by adding conditional jumps that are dependent on the result of opaque predicates.

Inserting dead code The term "dead code" refers to code blocks which are not or simply cannot be reached in the control flow graph and thus never get executed [49]. Inclusion of such code can make the analysis of a program more time consuming as it increases the amount of

code that has to be analyzed. For making the identification of dead code more difficult, opaque predicates that always resolve to either `true` or `false` can be used.

Inserting irrelevant code Cohen [47] described the concept of irrelevant ("garbage") code. Sequences of instructions that do not have an effect on the execution of a program can be inserted into the code in order to make analysis more complex. The most simple form of irrelevant code are NOP instructions which do not modify the program's state. In contrast to dead code, irrelevant code can be reached by the control flow of the program and gets executed at runtime, however, without having any effect on the program's state.

Reordering Similar to reordering of data structures, expressions and statements can be reordered as well to decrease locality in case the order does not affect the program behavior. While these techniques were originally introduced for code optimization [9], they are also applicable in an obfuscation context.

Loop transformations Many loop transformations have been designed to improve the performance and space usage of loops [9]. Some of them increase the complexity of the code and can therefore be used for obfuscation purposes. *Loop blocking* was originally designed to optimize the cache behavior of code. It breaks up the iteration space of a loop and creates inner loops that fit in the cache. In *loop unrolling*, originally developed to improve performance, the body of the loop is replicated one or more times to reduce the number of loop iterations. The *loop fission* method splits a loop into two or more loops with the same iteration space and spreads the loop body over these new loops.

Function splitting/recombination Function cloning describes the concept of splitting the control flow in two or more different paths that look different to the attacker, while they are in fact semantically equivalent. Another transformation type merges the bodies of two or more (similar) functions. The new method has a mixed parameter list of the merged functions and an extra parameter that selects the function body to be executed.

The related idea of overlapping functions – where the binary code of one function ends with bytes that also define the beginning of another function – is commonly used by compilers for optimization purposes and can also by used to confuse a disassembler. A similar but more sophisticated concept was introduced by Jacob et al. [106]. Two independent code blocks are interweaved in a way that, depending on the entry and exit points of the merged code, different functionality is executed.

Aliasing Inserting spurious aliases (i.e., pointers to memory locations) can make code analysis more complex as the number of possible ways for modifying a particular location in memory increases [102, 162]. These pointer-references can also be used as indirections to complicate the reconstruction of the control flow graph of a program in static analysis scenarios [202].

Name scrambling Modifying identifier names such as the ones of variables and methods and replacing them with random strings is a prime example for source code obfuscation, which is not covered in our work. While binary code usually does not contain identifier names any more, byte code preserves some of the identifier names. For example, Java byte code contains class, field, and method names. By substituting expressive names with random strings, semantic information that can be important for a human analyst is removed.

Control flow flattening This obfuscation method aims at removing the structure of a program's control flow graph. Wang et al. [201] described a concept called *chenxification* that puts the basic blocks of a program into a huge switch-statement (called dispatcher) which decides based on an opaque variable where to jump next in order to preserve the correct control flow. Control flow flattening using a central dispatcher was also described by Chow et al. [39]. A similar concept by Linn and Debray [130] uses a so-called *branch function* to obfuscate the targets of CALL instructions. All calls are forced to pass through the branch function which directs the control flow to the actual target based on a call table. Popov et al. [159] proposed the concept of replacing control transfer instructions by traps that cause signals. The signal handling code then performs the originally intended control flow transfer. Further control flattening techniques were described by László and Kiss [123] and Cappaert and Preneel [32]. Our concept for control flattening for software diversification is introduced in Section 3.

Parallelized code While originally being a code optimization technique, parallelizing code also got popular in the code obfuscation context as parallelized code is much more difficult to understand than sequential code [49]. Adding dummy processes to a program or parallelizing sequential code blocks that do not depend on each other increases the complexity of analysis [209].

Removing library calls The calling of libraries of the programming languages (particularly ones with a high level of abstraction) offers useful information to an attacker. Because they are called by their name, they cannot be obfuscated. By replacing standard libraries with own versions, these calls can be removed and thus their functionality obfuscated.

Breaking relations This technique aims at obfuscating relations between components of a program such as the structure of the calling graph or the inheritance structure of an object-oriented program. For example, classes can be split up (*factoring*); similarly, common features of independent classes that do not have common behavior can be moved into a new parent class (*false refactoring*).

2.2.3 Dynamic code rewriting

The main characteristic of code obfuscation schemes in this category is that the executed code differs from the code that is statically visible in the executable.

Packing/Encrypting Various malware obfuscation approaches analyzed in the literature follow the concept of *packing*, which hides malicious code by encoding or encrypting it as data that

cannot be interpreted by static analysis. Thus, an unpacking routine has to be used to turn this data back into machine-interpretable code. By changing the encryption/encoding keys, packed program code can easily be rewritten upon distribution in order to complicate simple pattern matching attacks (*polymorphism* [153]). *Metamorphism* is a concept that goes one step further and expands its mutation capabilities to the actual payload by applying various types of obfuscations such as function reordering or data structure modifications to the code before packing.

The concept of *packing* is also used for benign software. Both the reduction of storage requirements though compression and the aim for obfuscating the code of an application in order to deter program analysis are key motivations for the adaption of packing technologies. For these application areas, a large number of commercial packers such as VMProtect[1], ASPack[2], Armadillo[3], Execryptor[4], Enigma[5], PECompact[6], and Themida[7] as well as open-source tools (e.g., UPX[8] and Yoda[9]) exist. Most of these tools are also popular by malware authors to hide the maliciousness of their code [23]. Recently, Roundy and Miller [167] presented a survey on obfuscation techniques used in malware packers.

The concept of packing for software protection was also discussed in academia. Cappaert et al. [33] introduced a modified form of packing where code can be decrypted on a fine-granular basis right before execution using a key that is derived from other code sections. Wu et al. [210] introduced a polymorphism based concept called *mimimorphism* which modifies data as if it were code fragments. A taxonomy of packer based obfuscation schemes as well as similar techniques was presented by Mavrogiannopoulos et al. [141].

Dynamic code modification In this technique similar functions are obfuscated by providing a general template in memory that is patched right before its execution [49]. Static analysis techniques fail as the functionality is available at runtime only. Other concepts of dynamic code modification [109, 136] implement the idea of correcting intentionally erroneous code at runtime right before execution.

Environmental requirements Riordan and Schneier [163] proposed the concept of environmental key generation, in which a cryptographic key is not statically stored in a binary but constructed from *environmental* data collected from within the computing environment. Only if a specific environmental condition is met (called activation environment), the program is able to generate the key and execute its code. Outside the activation environment the program does not reveal its secrets to an attacker.

Similar concepts can be applied to code as well. Sharif et al. [179] proposed a malware obfuscation scheme that makes the code conditionally dependent on an external trigger value.

[1] http://vmpsoft.com (accessed April 04, 2014)
[2] http://www.aspack.com (accessed April 04, 2014)
[3] http://www.siliconrealms.com/armadillo.php (accessed April 04, 2014)
[4] http://www.strongbit.com/execryptor.asp (accessed April 04, 2014)
[5] http://enigmaprotector.com (accessed April 04, 2014)
[6] http://bitsum.com/pecompact (accessed April 04, 2014)
[7] http://www.oreans.com/themida.php (accessed April 04, 2014)
[8] http://upx.sourceforge.net (accessed April 04, 2014)
[9] http://yodap.sourceforge.net (accessed April 04, 2014)

Without knowledge of this specific value, the triggered behavior is concealed from dynamic code analysis. Similar techniques are widely used in malware.

Hardware-assisted code obfuscation Hardware tokens can be used to improve the strength of other code obfuscation techniques [19, 82, 217]. The basic idea is create a hardware-software binding by making the execution of the software dependent on some hardware token. Without this token, analysis of the software will fail, because important information (e.g. targets of indirect jumps) is not available.

Virtualization Virtualization describes the concept of converting the program's functionality into byte code for a custom virtual machine interpreter that is bundled with the program [116]. Virtualization can also be combined with *polymorphism* by implementing custom virtual machine interpreters and payloads for each instance of the program. Vrba et al. [199] proposed the combination of fine-granular encryption and virtualization to hide VM code from analysis. Collberg et al. [49] described a variant of this concept under the term *table interpretation*.

Anti-debugging and -disassembling techniques This obfuscation category includes techniques that actively oppose analysis attempts using disassembling or debugging. For example, attached debuggers can be detected based on timing and latency analysis or the identification of code modifications caused by software breakpoints. Another technique is the execution of undocumented instructions in order to confuse a code analysis tool or a human analyst.

2.3 Code Analysis

Aside from plain pattern matching approaches, we consider three broad classes of attacks against programs protected by obfuscation: automated static analysis, automated dynamic analysis, and manual reverse engineering by a human who has access to automated tools.

2.3.1 Static Analysis

Static analysis is widely used for optimizing code, finding or proving the absence of bugs, or generally for answering questions about programs. In its broadest sense, static analysis refers to any program analysis that is performed just by inspecting the code of a program of interest but without ever executing it on a real or virtual machine. Usually, however, the definition also demands that static analysis make a statement about *all possible executions* of the program. If the static analysis is *sound*, it guarantees to compute a property (an invariant) that holds on all possible executions of the program. Because of undecidability, static analysis can achieve this only by *over-approximating*, i.e., including behavior in its computation that can not occur in real executions of the program.

Static analysis is always performed with respect to some property of interest. A simple example is constant propagation, which is a standard analysis performed by compilers for optimization. It computes, for each program location, the variables that are guaranteed to contain a single constant value in all possible executions of the program. More precise static analyses can

over-approximate the set of memory states of a program to prove memory safety, or generally the absence of certain bugs. If such a safety proof fails, it can be either because the program is indeed unsafe (i.e., contains a bug), or because of the imprecision from over-approximation in the analysis (i.e., a false positive). A static analyzer has to trade off precision against cost. A coarse analysis such as constant propagation that computes invariants just for syntactic program locations terminates relatively quickly. Finer grained analyses that compute invariants at the level of individual program paths [66, 140] (i.e., path-sensitive analyses) become more expensive.

Static analysis is mostly performed on source code, but platforms such as JAKSTAB [113], BAP [24], or CODESURFER/X86 [10] provide the necessary infrastructure and abstractions to apply static analysis to binaries. Static analysis on binaries requires higher precision than analysis on source code. To just identify the possible control flow in a binary, it is necessary to over-approximate the possible sequences of program counter values with high accuracy [115].

From a conceptual standpoint, obfuscation decreases the precision a particular static analysis can achieve, i.e., obfuscation introduces additional sources of over-approximation [86]. For instance, as mentioned above control flow flattening forces all control flow to pass through a dispatcher. Because the dispatcher is executed many times on all paths, an invariant describing the program behavior at the location of the dispatcher will have to be very coarse. For instance, in the case of constant propagation, almost no variable will appear to be constant in the dispatcher, unless it is constant on all paths throughout the entire program. To maintain precision, the static analyzer would have to compensate by not maintaining just a single invariant per location, but by splitting invariants according to program paths.

If the details of a particular obfuscation technique are known, an analysis can be crafted to almost completely eliminate the effect of the obfuscation [112, 120]. But despite existing work on automated refinement of the precision of general static analysis [11, 13, 66, 100], tailoring analyzers for obfuscation schemes is still a largely manual process.

Besides the conceptual precision requirements, there are also significant practical challenges for statically analyzing obfuscated code. Existing tools always make some assumptions about the behavior or structure of the executable code, which can be broken by obfuscations. Many static analyzers fail even in presence of simpler obfuscations used regularly by malware [149]. Especially tools that depend on an initial disassembly phase, such as CODESURFER/X86 [10] or early semantic malware detectors [43, 114], are vulnerable to syntactic obfuscations. Such obfuscations target state-of-the-art disassemblers like IDA Pro that rely mostly on heuristics to discover all executable code [130]. Removing the separate disassembly phase and working on code directly improves resilience against these simpler obfuscation schemes [24, 115].

Nevertheless, especially dynamic obfuscations are difficult to handle for static analyzers. For example, no static tool to date is able to reliably deal with self-modifying code, even though one could, in theory, imagine the static analysis compensating by analyzing the possible runtime state of the code section. Our analysis in Section 2.4 considers static analysis tools at the current state-of-the-art, including minor adaptations for each particular scenario.

2.3.2 Dynamic Analysis

Dynamic analysis is used for observing the behavior of deployed and running systems, and it is today an important part of forensic analysis of malware [74]. Dynamic analyses are performed

over real executions of a program, either online (at runtime) or offline (over a recorded trace). Dynamic analysis is dual to static analysis: dynamic analysis considers just a subset of all possible executions of a program and is *under-approximate*. Therefore, dynamic analysis considers only real behavior but at the price of missing some behavior that can occur in real executions of the program.

Just like static analysis, dynamic analysis is also performed with respect to particular properties of interest. For example, a dynamic analysis may record all system call invocations, all executed instructions, or all data that is transmitted over the network. Any form of testing is dynamic analysis. Because of the under-approximation, each bug, warning, or suspicious behavior found with a test is real. But tests generally do not cover all possible behaviors of the program.

While static analysis trades off precision against cost, dynamic analysis trades off *coverage* against cost. Covering additional behavior requires re-executing the program. Exhaustively enumerating all possible inputs of a program is impossible, therefore a number of techniques have been proposed to separate program executions into equivalence classes of similar behavior and execute only a single trace for each class. Symbolic execution and dynamic test generation have been particularly successful [27, 87] in covering relevant inputs. In principle, these approaches can exhaustively enumerate all equivalence classes. The number of equivalence classes is generally infinite, however; therefore symbolic execution would iterate forever, just like refinement schemes in static analysis.

A significant advantage of dynamic over static approaches in the analysis of obfuscated code is that it can be applied to binaries with relative ease. In fact, it is often simpler to dynamically analyze binaries than source code, because traces recorded at runtime show addresses of instructions in the binary and not just (obfuscated) source code information. Though, the use of anti-debugging techniques can oppose dynamic analysis attempts.

Because dynamic analysis monitors what is actually executed at runtime, obfuscations cannot fully conceal the behavior of a program. For instance, a dynamic analysis can trace self-modifying code just like regular code if it records the opcode and operands of the current instruction in addition to the value of the program counter [189].

The main weakness of dynamic analysis is its inherent incompleteness. If a program exhibits a particular type of behavior only under very specific circumstances, it may never be observed by dynamic analysis. This is especially problematic for malware, which may respond only to certain "triggers" [58, 119]. In practice, systematic exploration methods such as symbolic execution are additionally limited by their underlying constraint solver which is used to identify valid control flow paths. A new control flow path that exhibits hitherto unseen behavior can only be triggered if the constraint solver is able to find an input that drives execution down that path. If the constraint includes clauses that lie outside the supported theory of the solver, such as non-linear or floating point constraints, or is simply too difficult, the symbolic execution engine is unable to cover the path. As a best effort solution, the engine can then resort to random testing (fuzzing) of input parameters [88].

As most dynamic analyses, symbolic execution-based techniques can also be applied to binaries to find vulnerabilities in compiled code. With the support of an emulator, the same methods can even be applied to kernel and device driver code [37, 183].

As for static analysis, Section 2.4 lists what is possible using the current state of dynamic

analysis tools, with only relatively minor technical adaptations.

2.3.3 Human Assisted Reverse Engineering

We use this very broad category to cover any kind of analysis that a skilled human reverse engineer can perform with the help of any state-of-the-art tool. The ability for creative problem solving and adaptation makes humans much more efficient in dealing with obfuscations than fully automated techniques. In contrast to a purely automated approach, however, humans can be mislead by clues that suggest structure, such as type names or inheritance relationships among classes.

Major improvements were archived for disassemblers and decompilers likewise throughout the past two decades, e.g. see [45, 174, 175]. The de facto industry standard for disassembling and reverse engineering, IDA Pro, now includes a powerful disassembler for the x86 architecture[10]. Academic research as well has made tremendous progress. The Boomerang Project[11] is an attempt to develop an open source decompiler. Early results from a study [76] conducted on a real-world program were promising. Despite the fact that only the core algorithm and not the entire program was decompiled, the experiment was able to demonstrate that decompilation with human assistance can be practical in certain use cases. The project, however, does not seem to receive much attention anymore. Recently, a new binary-to-C decompiler named PHOENIX was introduced by Schwartz et al. [174]. It implements a structural analysis algorithm that uses iterative refinement strategies as well as the property of semantics-preservation in order to archive significantly more accurate results than previous approaches.

A major driving force behind the development of human assisted code analysis approaches is the software cracking scene. In early years the SoftICE debugger has been a very popular tool among reverse engineers, however it is no longer maintained [207]. Today, the freely available debugger OllyDBG[12] to some extent continues where SoftICE left off. A large number of plug-ins that are dedicated to cracking and tampering purposes have been made available by the scene (e.g., Ollybone [185] for semi-automatic unpacking).

This code analysis category is naturally hard to grasp formally. In our analysis in the next section we base the capabilities of human reverse engineers on published results and common knowledge about the state-of-the-art.

2.4 Robustness Analysis

In this section we evaluate the effectiveness of different classes of code obfuscation schemes against attacks as described in the scenarios introduced in Section 2.1.3. The effectiveness of a specific type of code obfuscation is evaluated by comparing it to code analysis approaches described in the literature. Thereby, we assume that a developer can implement a particular obfuscation technique in maximum quantity, thus we evaluate the analyst's worst case scenario with the strongest possible obfuscator of one class. We are well aware of the arms race between

[10] https://www.hex-rays.com/products/decompiler (accessed April 04, 2014)

[11] http://boomerang.sourceforge.net (accessed April 04, 2014)

[12] http://www.ollydbg.de (accessed April 04, 2014)

code obfuscation and analysis and the fact that in theory every obfuscation technique can be broken with targeted analysis techniques (see Section 2.3). We focus on the status quo of code obfuscation in real-life application scenarios and evaluate the capabilities of state-of-the-art code analysis tools, also considering possible non-complex modifications to the analysis techniques in order to target particular obfuscations. The possibility of developing analysis techniques targeted to break a specific obfuscation scheme does not prove it useless in general as small modification to the obfuscation technique can again raise the bar for analysis.

As of today's knowledge, a precise formalization of the security of an obfuscation scheme seems to be difficult to achieve. Previous literature indeed describes concepts to quantify the hardness of a particular class of code obfuscation such as software complexity metrics [5, 49]. However, it remains unclear whether such notions are able to capture all security properties of the obfuscating transformation correctly. In this survey, we thus follow a different approach and rate the strength of each obfuscation class in a particular attack scenario based on three groups. A summary of the results can be found in Table 2.2. **Black** marks obfuscation techniques that break a certain type of program analysis fundamentally with respect to its state-of-the-art techniques. **Gray** defines scenarios in which a particular code obfuscation class cannot be considered as unbreakable, but still makes analysis substantially more expensive. Thus, while an analysis technique might work under lab conditions when dealing toy programs, limits of available resources could get reached in the analysis of larger programs. **White** marks obfuscation techniques that only result in minor increases of costs for analysis which thus cannot justify their application considering the limitations of the obfuscation (e.g., increased binary size and decreased runtime performance). Furthermore, we distinguish between ratings that are supported by results in the literature (marked with a checkmark in Table 2.2) and ones that are based on theoretical evaluation of the state-of-the-art.

2.4.1 Pattern Matching

The most basic form of pattern matching is limited to simple comparison of a program's code on byte level with some predefined pattern. Consequently, this type of pattern matching is weak against all kinds of code modifications, including obfuscating transformations. However, in the malware context several more sophisticated pattern matching concepts, mostly focussing on the identification of malicious behavior, were introduced in the literature and advanced pattern matching arguably can cope with naive obfuscation techniques such as equivalent instructions. In the following, we discuss the state-of-the-art in research on pattern matching based code analysis in the presence of different classes of code obfuscation schemes.

Locating data It is obvious that naive pattern matching techniques on data break as soon as the structure of the data changes. Thus, even simple obfuscations are effective against simple forms of pattern matching. This was confirmed by Moser et al. [149], who were able to show that simple data obfuscation techniques are sufficient to make pattern matching ineffective for the identification of data. The only exception are *anti-disassembling techniques* that interfere with static code analysis by forcing it into producing false disassembly and thus do not target pattern matching based analysis.

	PM		Autom. Static				Autom. Dynamic				Human Assisted			
Name	**LD**	**LC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**
Data obfuscation														
Reordering data			✓				✓							
Changing encodings	✓						✓							
Converting static data to procedures	✓						✓				✓			
Static code rewriting														
Replacing instructions		✓												
Opaque predicates				✓										
Inserting dead code						✓			✓	✓				
Inserting irrelevant code		✓												
Reordering														
Loop transformations														
Function splitting/recombination						✓								
Aliasing						✓			✓					
Control flow flattening	✓								✓	✓		✓		✓
Parallelized code														
Name scrambling						✓								
Removing standard library calls														✓
Breaking relations														
Dynamic code rewriting														
Packing/Encryption		✓		✓				✓	✓			✓		✓
Dynamic code modifications														
Environmental requirements														
Hardware-assisted code obfuscation											✓			
Virtualization			✓	✓				✓				✓		✓
Anti-debugging techniques						✓		✓	✓					✓

Legend		obfuscation breaks analysis fundamentally
		obfuscation is not unbreakable, but makes analysis more expensive
		obfuscation only results in minor increases of costs for analysis
	✓	A checkmark indicates that the rating is supported by results in the literature.
		Scenarios without a checkmark were classified based on theoretical evaluation.

Table 2.2: Analysis of the strength of code obfuscation classes in different attack scenarios (PM = Pattern Matching, LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

Locating Code Moser et al. [149] also demonstrated the limitations of pattern matching against the static code rewriting technique *control flow flattening*. Their reasoning can be easily extended to other forms of static rewriting and dynamic code rewriting as any modification of the binary clearly destroys the pattern.

In the literature, locating code through pattern matching is often described in a malware context. The primary aim of concepts in this attack scenario is the generation of generic patterns describing malicious behavior of program fragments in order to be able to automatically classify the maliciousness of software. While these approaches do not aim at understanding the semantics of the program, they still can identify locations in the code that implement abnormal (malicious) behavior. HANCOCK [91] is a system for the automated generation of signatures for malware. Through normalization of opcodes it is resistant against simple polymorphic code transformations such as register re-assignment. Christodorescu et al. [43] introduced the concept of semantics-aware malware detection, which employs sophisticated patterns (so-called templates) which describe functionality independently from its actual implementation. However, this approach is limited to simple replacement patterns. We therefore marked static code rewriting techniques in gray.

Dynamic code obfuscation techniques, for instance virtualization, remove the structure of the code entirely, thus rendering pattern matching based analysis approaches ineffective. To some extent, however, code analysis based on patterns is still possible as demonstrated in the literature. Tang and Chen [188] proposed the identification of polymorphic malware on a network stream using advanced statistical analysis. In contrast to fixed string pattern matching polymorphic versions of a program can be identified with this approach. POLYGRAPH [154] is a concept for the automated identification of polymorphic worms which exploits the existence of invariant substrings in all polymorphic variants of a malware. Dalla Preda et al. [63] generated signatures of metamorphic signatures through abstract interpretation of semantics developed for self-modifying code. All three approaches exploit the characteristic structure of polymorphic programs. To summarize, while it was shown in literature that pure identification of the obfuscation method *polymorphism* is feasible in a malware context, localization of particular functionality inside the binary is still impossible with pattern matching. As in the previous scenario, *anti-disassembling techniques* do not provide additional protection against pattern matching approaches aiming at locating code.

2.4.2 Static code analysis

Locating data Similar to pattern matching, automated static code analysis of data is limited when analyzed data is not stored in its original representation. In Table 2.2 the data obfuscation class *reordering* is marked in gray, because a simple data flow analysis, which is necessary for the reconstruction of reordered data, is arguably possible with automated static analysis tools such as JAKSTAB [113]. Furthermore, it can be argued that a static code rewriting technique can be effective against localization of data if it complicates the reconstruction of the control flow graph of the program: static data flow analysis strongly depends on knowledge on the control flow. For this reason, control flow modifying obfuscation techniques are marked in gray in Table 2.2.

Dynamic code rewriting in general is effective against localization of data inside programs as the original representation of the data is destroyed. At first sight, static analysis appears completely pointless against virtualization obfuscation as only the code of the interpreter can be directly analyzed. In static analysis this leads to an effect which Kinder [112] called *domain flattening*: data flow information from different locations in the original program are merged to one location in the interpreter, resulting in a more imprecise analysis. However, Kinder was able to demonstrate that by lifting the static analysis to a second dimension of location (the virtual program counter), the same analysis precision as on unobfuscated code is achievable. While the introduced approach was evaluated on a toy example only, the preliminary results still indicate promising directions for static analysis of VM-protected binaries. For that reason, we marked *virtualization* in gray for this analysis scenario. *Anti-disassembling techniques* can, to some extent, limit static analysis and thus are marked in gray as well.

Locating code The localization of a particular program feature in binary code through static analysis is mainly based on an analysis of the structure of the control flow graph (e.g., [97]) of the program. Therefore, code obfuscation schemes that modify or hide the control flow of a program (*opaque predicates*, *loop transformations*, *parallelized code*, etc.) can be considered as candidates for protection against static analysis techniques. *Opaque predicates* are the most important concept for control flow obfuscation in the presence of a static code analysis tool. Dalla Preda et al. [60] proposed an abstract interpretation-based methodology for removing simple *opaque predicates*. This automated, static concept was shown to be more complete than approaches for dynamic analysis of opaque predicates. However, as the authors state, their analysis concept is limited to simple types of opaque predicates only. Thus, we consider more complex opaque predicates still effective in making static reconstruction of the control flow graph significantly more difficult. Tsai et al. [195] introduced a framework for analyzing control flow obfuscation by representing it as a composition of atomic operators in order to evaluate robustness. Still, it remains unclear to what extent such theoretical results can support the evaluation of the strength of an obfuscation in real-life applications. We conclude that while no general statement regarding the strength of obfuscation can be made for this particular attack scenario, obfuscation schemes that complicate the reconstruction of the program's control flow graph can still make the attacker's aim of identifying the entry point into a particular functionality considerably harder.

Code obfuscation based on dynamic code rewriting makes static analysis considerably more difficult as the analyzed code in the binary does not correspond to the code that actually gets executed. However, in the research area of malware identification, approaches to circumvent obfuscation were introduced in recent literature. *Malware packers* were attacked with heuristics-based static analysis techniques (e.g. [194]) and comparison with previously-seen malware samples [104, 110]. Similar to pattern matching based approaches, in automated static analysis, the maliciousness of code is evaluated by identification of abnormal structures of the program code. The idea of using model checking for detecting malicious code was proposed by Kinder et al. [114]. Furthermore, static analysis of *polymorphism* as well as *metamorphism* was discussed in recent literature. Bruschi et al. [25] compared a normalized version of the control flow graph of a binary against CFGs of known malware in order to detect malicious behavior. In a similar concept, Walenstein et al. [200] normalized program code (e.g. through simple instruction

substitution patterns) to be able to compare it to known malware samples. Coogan et al. [53] used a combination of two static code analysis techniques (slicing and alias analysis) to identify malware packers. The idea of using the frequency of opcodes as a predictor for malware was proposed by Bilar [16]. For example, massive use of mathematical operations might indicate malware that tries to obfuscate its malicious behavior using a packer based approach. Furthermore, it was shown that a high frequency of usually rare opcode is an indicator for polymorphic or metamorphic malware. Chouchane and Lakhotia [38] proposed the detection of metamorphic malware by creating signatures for instruction-substitution engines. However, similar to pattern matching based approaches, none of the introduced concepts for static analysis is able to find the location of specific functionality. Only the characteristic structure of a malware packer can be identified. As a consequence, we marked *packing/encryption* in gray. Like in the previous scenario (locating data) *virtualization-obfuscation* was marked gray in Table 2.2 because of promising preliminary results in Kinder's discussion [112] on the challenges of statically analyzing virtualization-obfuscated programs.

Extracting code While in software engineering concepts for reusing functionality from legacy binary code exist (e.g., [182]), the automated static extraction of obfuscated code has not been widely discussed in the literature.

Our reasoning regarding the effectiveness of different classes of code obfuscation techniques in this attack scenario is simple: extracting code from a program through static analysis is at least as difficult as locating code because the latter is a fundamental requirement for the former. Table 2.2 indicates differences between code localization and extraction in the scenario of static code rewriting for four obfuscation techniques. *Aliasing*, *control flow flattening*, *parallelized code*, and *breaking relations* share the common effect of increasing code dependencies by interweaving independent parts of the program. These interweavings are difficult to resolve through static analysis, thus making the extraction of code sections considerably harder than its pure localization. Collberg et al. [49] first described the effect of raising analysis complexity when extending the scope of the obfuscating transformation. Analogously, in the category of dynamic code rewriting *virtualization* was marked in black compared to gray in the locating code scenario because of the interweavings that make it difficult to extract all required code sections. *Anti-debugging and -disassembling* was marked in black due to the existence of a wide range of techniques that actively interfere with static disassembling [21].

Understanding Code In order to allow a human analyst to gain a better understanding of an obfuscated program an automated analysis tool has to be able to remove at least parts of the applied code obfuscation scheme from the binary.

Several concepts for static de-obfuscation with the aim for making the program code more understandable for a human analyst were proposed in the literature. An early work by Rugaber et al. [169] has shown that detecting interleaved code (e.g., through *function recombination*) is a time consuming task. Majumdar et al. [139] evaluated the robustness of the obfuscation technique *aliasing*, where two or more pointers refer to the same memory location. An experimental evaluation showed that resilience expected from the theoretical approach does not hold in real-life scenarios; still, in general it is difficult to evaluate the actual strength of *aliasing*. Guillot

and Gazet [95] developed techniques for automated static de-obfuscation. The basic concept is to make program code more understandable by automated rewritings based on local semantic analysis, similar to optimization steps made by compilers. Raber and Laspe [161] introduced a plugin for IDA Pro that is capable of removing basic obfuscation and anti-debugging techniques from a binary. A shared limitation of all introduced concepts is the major gap in success rates between academic examples and real-world scenarios. While these concepts show that theoretical analysis models work under laboratory conditions, practical application is limited. Thus, we marked data obfuscation as well as static code rewriting techniques in gray.

In Table 2.2 *name scrambling* was marked in black, because an identifier's semantics is of fundamental importance for a human's understanding of a program, but cannot be restored with the help of automated code analysis techniques as the transformation is one-way [49]. *Parallelized code* can implicitly be considered as a strong obfuscation technique in this analysis scenario as code extraction, which is a fundamental requirement for code understanding, is also difficult. Following the same line of reasoning dynamic code rewriting methods in general are strong in the context of code understanding through automated static analysis.

2.4.3 Dynamic code analysis

Locating data Every known data obfuscation technique except for white-box cryptography has a limited practical effect in dynamic attack scenarios as data is always visible to an attacker at some point during runtime. For example, although the cryptographic key of a DRM client might be stored in an obfuscated way (e.g. through data encoding) in the binary, during runtime the cryptographic algorithm has to reconstruct the original representation of the key in order to perform decryption tasks. In literature, several concepts for an automated dynamic extraction of data structures from program binaries were proposed. Zhao et al. [216] introduced a concept for dynamic extraction of data in malware. Cozzie et al. [56] extracted data structures from memory dumps using Bayesian unsupervised learning. Lin et al. [127] introduced a system called REWARDS which reveals data structures through observation of the program execution. It marks each memory location that was accessed at runtime with a timestamp and traces the propagation of data. A similar concept called HOWARD was proposed by Slowinska et al. [180] in 2011. It aims at extracting data structures from binaries by dynamically tracing (using QEMU-based emulation) how a program accesses the memory. HOWARD is able to reconstruct large parts of the symbol table. It thus simplifies the progress of reverse engineering and improves readability of obfuscated code as well as data.

To sum up, in a dynamic analysis context, most static as well as dynamic code rewriting techniques do not provide significant additional security in the context of data protection. The only exception are obfuscations that require special runtime enablers (additional hardware or environmental conditions) to execute. These techniques can withstand dynamic analysis in situations where the runtime enabler is not present. For this reason, both the techniques *environmental requirements* and *hardware-assisted code obfuscation* were marked in gray in Table 2.2.

Locating code Localizing a particular feature inside binary code through dynamic analysis is based on the observation of the program's behavior. Static code rewriting techniques are not

effective in dynamic attack scenarios due to one important factor: most of them are not explicitly targeted at the prevention of dynamic analysis. *Replacing instructions*, *inserting dead code*, *opaque predicates*, *code insertions*, etc. were in the first place developed to obfuscate the semantics of binary code. However, automated dynamic analysis techniques depend on these semantics to a much smaller extent than static analysis or even human analysis, thus are less affected by the obfuscation. In the literature very diverse concepts for dynamic analysis of obfuscated code were proposed. Li et al. [126] described a technique that identifies malicious behavior based on the malware's runtime system call sequences. McVeto [189] is a model checker for machine code. While traditional dynamic analysis suffers from the problem of incompleteness as a program's behavior can only be analyzed on one input at at time, McVeto implements a combined static and dynamic approach which aims at reaching more code locations by actively manipulating program inputs. Another strategy for finding a particular feature in program code was discussed by Deprez and Lakhotia [72] and Wilde and Scully [206]. The basic idea is to execute the program twice with two different inputs whereby one input invokes the feature and the other does not. From calculating the differences between the two traces conclusions on the location of the particular feature can be drawn. Madou et al. [134] discussed the effectiveness of hybrid (static and dynamic) analysis approaches and demonstrated it in the context of the reconstruction of an obfuscated control flow graph (*control flow flattening*).

While dynamic code analysis is strong against static code obfuscation, dynamic obfuscation techniques are much more robust against it due to their ability to dynamically modify the code and thereby making program traces look different for each run. However, attacks against dynamic code rewriting obfuscation exist as well and can primarily be found in the malware analysis context. In recent literature several strategies for the automated dynamic analysis of packed programs were proposed. Moser et al. [148] proposed a solution for the incompleteness problem of dynamic analysis by making the exploration of multiple execution paths possible. Thus, the approach allows the identification of malicious behavior that is executed only if a certain conditions is met. REANIMATOR [52] is two-step malware identification system. First, known malware is dynamically analyzed and code that is responsible for a particular malicious behavior is modeled. This model can then be used in a second step to identify the same malicious behavior in other code samples through static analysis. However, dynamic code rewriting techniques such as packing can limit the detection rate of this approach significantly. Sharif et al. [177] introduced the EUREKA framework that automatically extracts the payload of a packed program by running the binary in a virtual machine. RENOVO by Kang et al. [108] is another dynamic unpacker which employs monitoring of executed instructions and memory writes at runtime for the extraction of a hidden payload. It is based on the dynamic code analysis component TEMU [214] of the BITBLAZE platform [183]. Heuristics- and statistics-based strategies are used to determine the exact moment when the unpacking process is finished and the unpacked code is fully stored in memory. Royal et al. [168] introduced a behavioral based approach for automated unpacking inside a VM. Its combination of static and dynamic analysis identifies unpacking routines through its characteristic behavior. A malware analysis approach by Debray and Patel [70] focusses on the automated identification of unpacking routines inside binaries. Gröbert et al. [92] proposed the detection of cryptographic algorithms by analyzing program execution traces, which show unique characteristics depending on the implemented algorithm.

In a similar concept named ALIGOT [28], the identification of cryptographic algorithms in execution traces is based on the comparison of input-output relationships with known cryptographic algorithms. With this concept even heavily obfuscated algorithms can be identified because the input-output relationship does not differ from the original version of the algorithm.

Furthermore, several approaches for automated dynamic analysis of programs protected by virtualization were introduced in recent years. Sharif et al. [178] proposed the use of tainting and data-flow analysis techniques in order to find the byte code implementing the payload of the virtualized program. The described automated reverse engineering approach is able to reconstruct control flow graphs and was evaluated against the code virtualization tools VMProtect and Code Virtualizer. A different strategy for automated dynamic analysis of virtualized code was proposed by Coogan et al. [54]. Instructions that contribute to arguments of system calls are collected in order to get an understanding of the functionality of the program. Webster and Malcolm [204] proposed the use of formal algebraic specifications to detect metamorphic and virtualization-based malware.

To conclude, we marked dynamic code rewriting approaches in gray because practical application of all described approaches is limited to malware identification tools only. In other words, they do not directly aim at locating particular functionality, but malicious behavior in general.

Extracting code Similar to the scenario of static extraction of code sections, dynamic code extractors have to deal with dependencies between different parts of the program. Several static classes of code obfuscation add bogus dependencies in order to make analysis more difficult. Thus, similar assumptions on the resilience of the code obfuscation techniques can be made. Still, several dynamic concepts for automated extraction of code section were described in the literature. TOP by Zeng et al. [215] collects instruction traces and translates the executed instructions into a high level program representation which can be reused as a normal C function in new software. The authors claim the concept to be resilient against the obfuscation techniques *packing/encryption*, *aliasing*, *control flow obfuscation* (e.g., *flattening*), *inserting dead code*, as well as several popular *anti-debugging techniques*. Following the results of Zeng et al. [215] we also marked *parallelized code*, which is arguably less effective in dynamic code analysis scenarios, in gray.

Most other concepts introduced in recent literature are focused on malware detection. Leder et al. [125] proposed a concept for automated extraction of cryptographic routines through dynamic data analysis. The automated isolation of a single function from a (malicous) binary was proposed by Caballero et al. [26]. In contrast, INSPECTOR by Kolbitsch et al. [118] allows the automated extraction of a particular malicious behavior that does not necessarily have to be limited to one function of the program only. Following the evaluation of this approach, the dynamic obfuscation class of *packing/encryption* has to be considered as weak in the presence of the described approach.

Other dynamic code rewriting techniques were marked in gray in Table 2.2. In contrast to the locating code scenario we marked *virtualization* in black as Sharif et al's concept [178] for analyzing virtualization-protected code is limited to the localization of code and not its extraction.

Understanding code Analogously to previous scenarios, getting a deeper understanding of a program's code requires at least basic de-obfuscation functionality in the automated analysis approach. In literature, several concepts were introduced. Udupa et al. [197] proposed automated de-obfuscation of *control flow flattening* and *dead code* insertion using a hybrid approach of static and dynamic analysis techniques. The incomplete control flow graph from a dynamic analysis is enriched by adding some control flow edges that could possibly be taken through static analysis. While the results presented in the paper show that the evaluation of isolated analysis problems is possible, it is difficult to reason on the value of the concept for real-life programs.

2.4.4 Human analysis

The capabilities of a human code analyst is difficult to quantify in general. Tilley et al. [190] first described a framework for program understanding including cognitive aspects of a human reverse engineer. However, it is still safe to assume that provided with sufficient patience a human analyst can break any class of code obfuscation and is only limited by scalability constraints.

Locating data Most data obfuscation techniques have only limited strength in the attack scenario of a human analyst trying to locate data structures in programs. One important concept for the protection of data inside a binary is *white-box cryptography*. It was proposed to prevent the extraction of a cryptographic key from the binary by mixing it with the algorithm. A decade ago, the first implementations of white-box algorithms for DES and AES have been proposed [22, 40, 41, 128]. However, all of them have been broken using techniques such as fault injection [105], statistical analysis [129], condensed implementation [212], differential cryptanalysis [17, 68, 90, 213] or generic cryptanalysis [146]. Given this mixed history, in recent years, research on white-box cryptography focused on the question how the general idea and its security concepts can be backed by a theoretical foundation. Wyseur [211] discussed the state-of-the-art of *white-box cryptography* and proposed new block ciphers and design principles for the construction of white-box cryptographic algorithms. Saxena et al. [170] described a theoretical model of *white-box cryptography* using appropriate security notions and presented both positive and negative results on white-box cryptography. This leads us to the conclusion that in its current state the strength of *white-box cryptography* is unproven, although it can make the extraction of the cryptographic key considerably more complex. Thus, we marked the obfuscation class *converting static data to procedures* in gray. Other obfuscation techniques that can provide at least limited resilience in this attack scenario are *environmental requirements* and *hardware-assisted code obfuscation* because of their dependencies on external factors such as the presence of a particular hardware token. The strength of dongles for software protection was evaluated by Piazzalunga et al. [158]. The authors developed a model for forecasting the amount of time an attacker would need to break dongle based software protection schemes and concluded that today's available dongle solutions provide only minimal protection. Still, we marked the class of *hardware-assisted code obfuscation* techniques in gray as dongles are only the most basic approach, while several more sophisticated concepts were introduced in the literature [19, 82, 217] that are more resilient to dynamic analysis concepts.

Locating Code Some obfuscation techniques break abstractions of a program's code which would be important for human understanding. *Name scrambling*, *removing standard library calls*, and *breaking relations* do not prevent an automated tool from analyzing a program. However, they can make manual analysis by a human more difficult.

Other classes of code obfuscation have limited robustness against a human analyst trying to locate code. For example, Rolles [165] introduced a semi-automated de-obfuscation approach against virtualization-obfuscation, which is based on reverse engineering the virtual machine, extracting the byte code and then turning it into native code. Malou et al. [137, 138] developed an interactive de-obfuscation tool named LOCO that allows an analyst to navigate through the control flow graph of a program in order to undo static obfuscating transformations such as *control flow flattening*. Quist and Liebrock [160] demonstrated how a sophisticated visual representation of the control flow of a program can speed up the analysis process. In particular, unpacking routines of malware can be identified efficiently using a visualization approach.

Extracting Code Finding the location of code is a prerequisite for code extraction. Thus, similar to the scenario of dynamic analysis, the extraction of code can be considered at least as difficult as the localization of code for a human attacker. One of the most popular approaches for human-assisted code extraction is program slicing which is based on the idea of reducing the program's code to a minimum (a so-called *slice*) that still produces a particular behavior (i.e., a subset of instructions that can affect the value of a variable of the program). In the literature, a multitude of program slicing approaches have been introduced throughout the past two decades (e.g., [121] and [155]). A major limitation of program slicing is that the human analyst needs to have a deep understanding of the program's internals in order to be able to specify a slicing criterion (e.g., relevant variables and behavior) on a code level. Several attempts have been made to raise the level of abstraction in slicing and thus making it less dependent on manual analysis such as conditioned slicing [30, 31, 46, 64] and constraint slicing [65, 79, 80]. We marked *opaque predicates* and *aliasing* in gray as these obfuscation techniques can make the identification of the minimal subset that still implements a particular functionality more difficult.

Understanding Code Despite the fact that getting a full understanding of a program can be considered as the most ambitious aim of a human analyst, today's state-of-the-art in code obfuscation provides only limited protection in this attack scenario. This assumption is backed by a plethora of reports about successfully removed copy protection schemes in the context of digital media such as CSS (DVD copy protection) or Windows Media DRM [152] in the past and HDCP (high-bandwidth digital content protection) [131] in recent years. Another piece of evidence for this assumption is the successful reverse-engineering of the VoIP software Skype. While Skype is known for its massive use of code obfuscation, Biondi and Desclaux [18] still were able to reveal the internal structure of the software. Furthermore, the client software of the cloud service provider Dropbox was successfully reverse-engineered despite being heavily obfuscated [111]. A decisive factor for these recent success stories in code analysis are today's sophisticated reverse engineering tools such as the industry standard IDA Pro that have become better and better in dealing with obfuscated code [73, 75, 77]. It can be concluded that almost all code obfuscation techniques have to be considered as ineffective in the presence of a human an-

alyst that puts enough time and effort into manual de-obfuscation. We marked *name scrambling*, *removing standard library calls*, *breaking relations*, and *virtualization* in gray as these obfuscations can make manual analysis by a human more difficult. Furthermore, the obfuscation classes *environmental requirements* and *hardware-assisted code obfuscation* can be considered as strong against human analysis as long as the external requirement cannot be accessed by the analyst. *Anti-debugging and -disassembling* was also marked in gray as all human-assisted analysis approaches described in the literature are still based on automated static and dynamic analysis tools.

CHAPTER 3

Code Obfuscation through Diversification of the Control Flow Graph

In this chapter we introduce a novel code obfuscation technique that effectively prevents static reverse engineering and limits the impact of dynamic analysis. Technically, we apply the concept of code diversification to enhance the complexity of the software to be analyzed. Diversification is used to prevent "class breaks", so that a crack developed for one instance of a program will most likely not run on another instance and thus each copy of the software needs to be attacked independently. While previous approaches required each diversified instance of a program to be unique on binary layer, we introduce the novel concept of control flow graph diversification, which mitigates this restriction.

3.1 Approach

Our approach combines obfuscation techniques against static and dynamic reverse engineering. As discussed in Chapter 2, static analysis refers to the process of automated reverse engineering of software without actually executing it. Using a disassembler, an attacker can translate machine code into assembly language, a process that makes machine instructions visible, including ones that modify the control flow such as jumps and calls. This way, the control flow graph of the software can be reconstructed without executing even a single line of code. By inserting indirect jumps that do not reveal their jump target until runtime and utilizing the concept of a branching function we make static control flow reconstruction more difficult.

Employing code obfuscation to prevent static analysis is a first step towards running code securely, even in the presence of attackers who have full access to the host. However, an attacker is still able to perform dynamic analysis of the software by executing it. The process of disassembling and stepping through the code reveals much of its internal structure, even if obfuscating

transformations were applied to the code. Preventing dynamic analysis in a software-only approach is not fully possible as an attacker can always record executed instructions, the program's memory, and register values of a single run of the software. However, in our approach we aim at making dynamic analysis considerably harder for the attacker by applying concepts from diversification. In particular, the information an attacker can retrieve from the analysis of a single run of the program with certain inputs is useless for understanding the trace of another input. It thus increases costs for an attacker dramatically, as the attacker needs to run the program many times and collect all information to obtain a complete view of the program. This concept can be considered as diversification of the control flow graph.

3.1.1 Protection against Static Reverse Engineering

In our approach we borrow the idea of a branching function to statically obfuscate the control flow of the software. While previous implementations replace existing `CALL` instructions with jumps to the branching function, we split the code into small portions that implement only a few instructions and then jump back to the branching function. While this increases the overhead, it makes the blocks far more complex to understand. Because of the small size of code blocks, they leak only little information: A single code block usually is too small for an attacker to extract useful data without knowing the context the code block is used inside the software. The jump from the branching function to the following code block is indirect, i.e. it does not statically specify the memory address of the jump target, but rather specifies where the jump target's address is located at runtime. Static disassembling results in a huge collection of small code blocks without the information on how to combine them in the correct order to form a valid piece of software.

Figure 3.1 explains this approach. The assembly code of the software is split into small pieces, which we call *gadgets*. At the end of each gadget we add a jump back to the branching function. At runtime, this function calculates, based on the previously executed gadget, the virtual memory address of the following gadget and jumps there. The calculation of the next jump target should not solely depend on the current gadget, but also on the history of executed gadgets so that without knowing every predecessor of a gadget, an attacker is not able to calculate the address of the following one. We achieve this requirement by assigning a signature to each gadget (see Section 3.1.3). During runtime, the signatures of executed gadgets are summed up and this sum is used inside the branching function as input parameter for a lookup table that contains the address of the subsequent gadget. Without knowing the signature sum of all predecessors of a gadget, it is hard to calculate the subsequently executed gadget.

3.1.2 Protection against Dynamic Reverse Engineering

The approach effectively prevents static analysis, as a debugger is not able to connect gadgets to each other without calculating signature sums and executing the branching function. Dynamic analysis, however, reveals all gadgets used in a single invocation of the software as well as their order. An attacker can easily remove the jumps to the branching function by just concatenating called gadgets in their correct order. By performing this task for several inputs, he gets significant information on the software behavior.

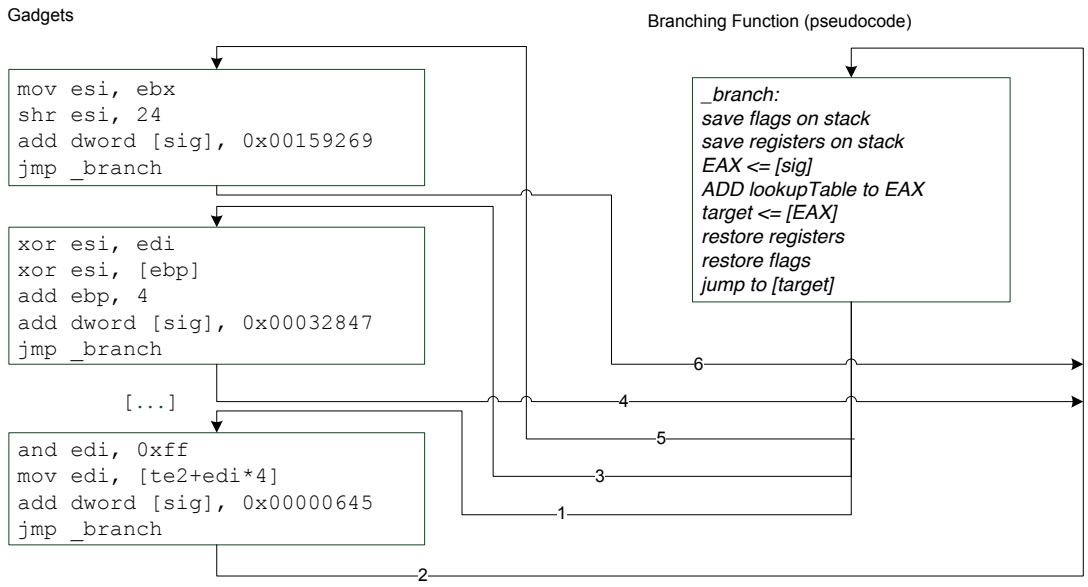

Figure 3.1: Overall architecture of the obfuscated program: small code blocks (gadgets) are connected by a branching function.

To mitigate that risk, we diversify the control flow graph of the software so that it contains many more control flow paths than the original implementation. We diversify gadgets (i.e. add semantically identical but syntactical different gadgets to the code) and add input dependent branches so that different gadgets get executed upon running the software with different inputs. We can symbolize this by a gadget graph, where the actual gadget code is stored in the edges that connect two nodes, which symbolize the state of a program. Figure 3.2 shows the multi-target branching concept before gadget diversification. For every node, we create outgoing edges and fill them with gadgets (i.e. instructions from the original code). All outgoing edges of one node start with the same instruction and only differ in gadget length. In a further step, these gadgets are diversified. Every path through the graph is a valid trace of the program. The branches are input dependent: based on the program's input the branching function decides which path through the graph has to be taken. For a logical connection between gadgets, we implement a path signature algorithm that uniquely identifies the currently executed node and all its predecessors (see Section 3.1.3).

In order to increase the security of the obfuscation, we prevent that a path that is valid for one input is also valid for other inputs. We do this by modifying some instruction's operands and automatically compensate these modifications during runtime by corrective input data. Consider, for example, the assembly instruction `add eax, 8`. If we replace this instruction with `add eax, ebx; sub eax, 1`, where the content of the register `eax` is derived from the program's input, only a value of 9 in `ebx` would yield to the correct value in register `eax`. Figure 3.3 shows a more complex control flow graph.

All paths through this graph are valid and semantically equal traces of the program. How-

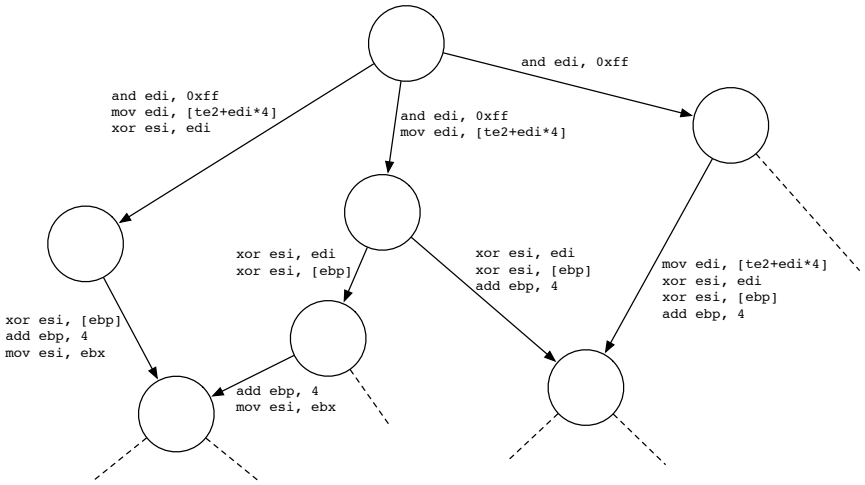

Figure 3.2: Gadget graph.

ever, because of the inserted modifications to operands, one specific path yields correct computation only for a specific input (or a group of inputs) and fails otherwise. If an attacker would use the trace of one input for running the program in the context of another input (e.g. by diverting the control flow in the branching function), our modifications to operands would not be compensated by the new input and the program would show unexpected behavior and might crash at some point (e.g. because of access to miscalculated memory addresses). The process of creating the diversified gadget graph is much easier and faster than breaking the obfuscation as an attacker has to obtain each trace individually.

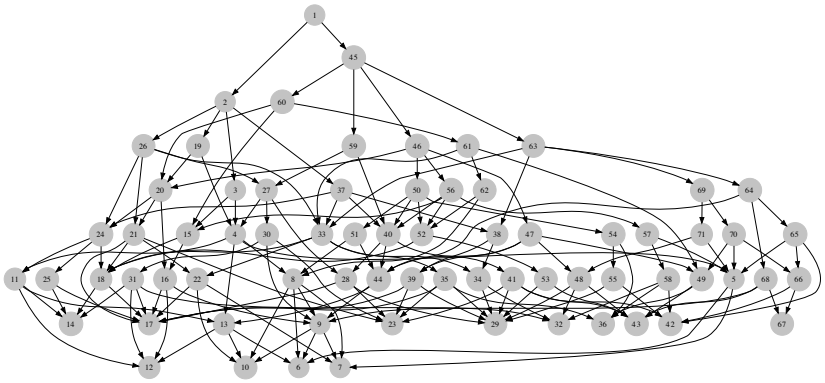

Figure 3.3: Diversified control flow graph.

At the beginning of our obfuscation algorithm, a random gadget graph is created from the software to be obfuscated, based on the input parameters for branching level and gadget size. We then generate unique path signatures (for details see Section 3.1.3) inside a depth-first search that traverses through all possible paths of the graph. Furthermore, we diversify the gadget code

(see Section 3.1.4), assign the path signature to the gadget and add the gadget to the output file. For every possible path that can be taken to reach a gadget, we add the gadget's memory address and path signature sum to the lookup table. Finally, we attach the branching function and the lookup table to the obfuscated code. Algorithm 3.1 shows the obfuscation algorithm in pseudocode.

Algorithm 3.1: Obfuscation algorithm in pseudocode

```
1  create random gadget graph;
2  forall the possible paths do
3      while path signature of current gadget is not unique do
4          create random path signature;
5      diversify gadget code;
6      add path signature to gadget;
7      output gadget code;
8      add gadgets memory address and path signature sum to lookup table;
9  output branching function;
10 output lookup table;
```

3.1.3 Graph construction

The main challenge of our approach against dynamic reverse engineering is the performance of the obfuscation algorithm. One the one hand, our approach aims to significantly delay dynamic analysis of an attacker by making it hard to traverse the entire graph within a reasonable time frame (i.e. a brute force attack). However, on the other hand, the initial construction of the graph has to be dramatically less time consuming than an attack. We solve this problem with full knowledge of the structure of the graph at obfuscation time compared to runtime. The obfuscation algorithm creates the graph and stores its structure in memory, allowing very efficient graph traversal at obfuscation time. In contrast, an attacker only has access to the binary code of the software that does not contain an explicit description of the graph's structure. An attacker has to execute all (or at least most) paths of the graph through the branching function, including the gadget's entire code, in order to rebuild the graph and obtain a complete view of the software.

Our graph construction algorithm takes the original program code as well as a minimum and maximum gadget size and a minimum and maximum branching size as input parameters and is based on a depth-first search. Starting at the root node, the algorithm adds a random number of child nodes (within the bounds of the branching size) and assigns a gadget to each connecting edge. All edges to child nodes contain the same code by means of being filled with a random number of instructions (within the given bounds on the gadget size) from the original code. Only the gadget size and therefore the number of instructions differ at this stage. Gadgets are not diversified at graph construction time. We define the absolute number of instructions executed until reaching a node of the graph as *node level*. Before adding a new node to the graph, the algorithm calculates the node level of the new node and checks if it already exists

anywhere in the graph. It that case, instead of creating the node, the algorithm links to the existing node. This method prevents a continually growing width of the graph.

During gadget graph construction, we calculate and store a path signature in each node. We make it unique (see below) so that it clearly identifies the node and all its predecessors. The signature is based on simple `ADD` and `SUB` assembly instructions on a fixed memory location. Each gadget adds (or subtracts) a random value to (or from) the value stored in memory. When traversing through the graph, the value stored at the memory location identifies the currently executed gadget and the path that was taken through the graph to reach this gadget. A node can have more than one signature, as more than one path of the graph could reach this node. In that case, each node signature uniquely identifies one of the possible paths from the root to the node. During signature assignment we prevent collisions (two nodes sharing the same signature), by comparing the current signature to all previously calculated signatures and choosing a different value for the `ADD` or `SUB` instruction if needed. We decided to implement a trail-and-error approach instead of an algorithm that generates provable distinct signatures to avoid performances bottlenecks at runtime. Figure 3.4 shows the path signature for a small graph.

We further add a second input parameter to the branching function described in the static part of our approach. Now, both the program's input and the path signature are input parameters for a lookup table that determines the next gadget to be called. To eliminate any information leakage from the branching function's input value, only a hash value of the program's input and the path signature is stored in the lookup table.

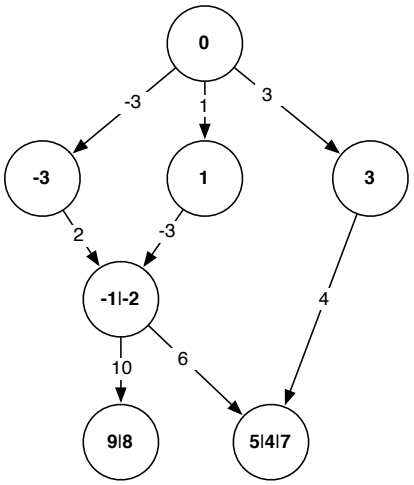

Figure 3.4: Path signatures.

3.1.4 Automatic Gadget Diversification

An efficient generation of semantically equivalent mutations of gadgets is the key challenge for software diversification. This process has to be fully automatic to be able to process large amounts of source code and the transformation function is preferably one-way to prevent differ-

ential analysis of gadgets. Pattern-based diversification algorithms (e.g. [69]) are a reasonable first code replacement step. However, the fact that an attacker only has local view on a gadget, can help to improve the strength of the diversification by inserting code dependency problems that are locally undecidable for an attacker.

We propose a combination of dummy code insertions and a process we call *instruction splitting*. The idea is to split basic instructions into two ore more instructions that are in combination semantically equivalent to the original instruction and then insert dummy code instructions in between them. We create bogus dependencies between the actual gadget code and dummy instructions by accessing data of split instructions inside the dummy code. To identify and remove dummy instructions, an attacker has to be sure that the code does not perform any vital operations on the code that is executed afterwards. However, this problem is hard to decide due to dependencies between gadgets. Because of the small gadgets sizes, an attacker only has local view on a gadget without knowledge of the subsequently executed gadget.

A simple example is the instruction `add eax, 5` that can be split into the two instructions `add eax, 2` and `add eax, 3`. Of course, this simple transformation provides only very limited security against automatic gadget matching algorithms. We can, however, tremendously improve the strength of the transformation by inserting dummy code. For example, the instruction `mov dword [0x0040EA00], eax` can be considered as dummy code, if the value that is stored in `0x0040EA00` is not used anywhere later in the software. The instruction sequence `add eax, 2; mov dword [0x0040EA00], eax; add eax, 3` is only semantically equivalent to `add eax, 5`, if `mov dword [0x0040EA00], eax` is dummy code. For an attacker with only local knowledge, this is an ambiguous problem.

Simple pattern based transformations do not withstand automated attacks aiming at reversing the diversification. The instructions `test eax, eax` and `cmp eax, 0` are semantically equivalent, but the transformation is weak, because a very simple matching algorithm can easily identify them as equivalent. However, analogous to the instruction splitting method, multi-instruction patterns can be combined with dummy code insertions to enable strong diversification. To provide an example, consider the instructions `push ebp; mov ebp, esp`. A semantically equivalent expression would be `push ebp; push esp; pop ebp`. A simple substitution transformation of one version for the other would most likely not withstand an automated attack. However, if the transformation is combined with dummy code insertion (e.g. `push ebp; push esp; add esp, [0x0040EA00]; pop ebp`, where `0x0040EA00` is `0`), an attacker with local knowledge of the gadget can not reveal the dummy code instructions and hence can not decide gadget equivalence locally.

Listing 3.1 shows the transformation of an example code block. The transformation function τ adds dummy code (lines 4 and 6) and modifies the instruction `add ebp, 4` so that it only provides the correct functionality if the corresponding input `8` is loaded into register `eax`. This modification prevents an attacker from extracting this specific (and fully functional) trace and using it with other inputs. To be able to generalize a trace, all input dependent operand modifications would have to be removed, thus the entire code would have to be analyzed instruction by instruction.

```
XOR ESI, [EBP]
ADD EBP, 4
ADD EBX, 4
MOV EAX, [ESP+4]
JMP _branch

    ⇓

XOR ESI, [EBP]
SUB EBP, EAX
ADD EBP, 12
ADD EAX, 5
ADD EBX, 2
MOV DWORD [0x0040EA00], EBX
ADD EBX, 2
MOV EAX, [ESP+4]
JMP _branch
```

Listing 3.1: Example code block diversification and obfuscation.

3.2 Discussion

The following section discusses the impact of our obfuscation scheme on performance and size of the resulting program and evaluates security aspects.

Performance and Size. To demonstrate the effectiveness of our approach, we implemented a prototype that reads assembly source code and generates an obfuscated version of it. We measured the performance losses of a simple benchmarking tool as well as a standard AES implementation using 8 different gadgets sizes. While the dynamic part of our approach accounts for an increase in required memory space because of diversified copies of gadgets, execution time heavily depends on the size and implementation of the branching function, as it inserts additional instructions. The performance decreases with the number of gadgets, due to calls to the branching function, which are required to switch between gadgets. In contrast, the strength of the obfuscation is directly proportional to the number of gadgets, so a trade-off between obfuscation strength and performance has to be made. We compared different gadget sizes from 1 to 50 with the execution times of the non-obfuscated programs (see Figure 3.5). While very small gadgets result in significant performance decreases, the execution time for a program with a gadget sizes of 10 and bigger approximates the execution time for the original program.

Security. We classified our method with Collberg's metric. Potency (strength against humans) can be evaluated with software complexity metrics. *Program Length* [96], *Nesting Complexity* [98], and *Data Flow Complexity* [157] are increased by our obfuscating transformation and we rate its potency level similar to Collberg's transformation "Parallelize Code" (potency level:

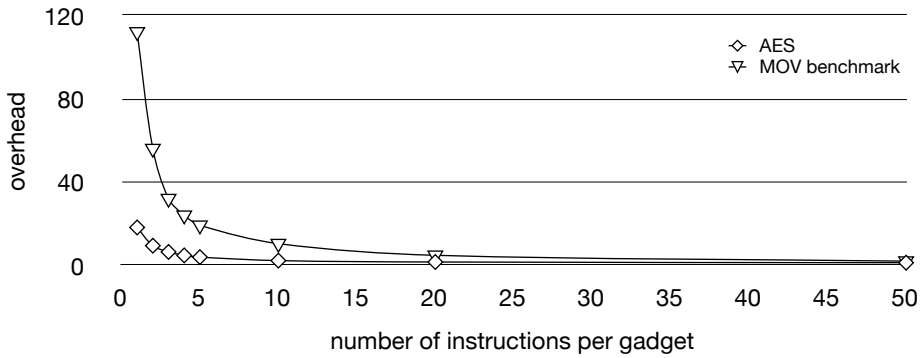

Figure 3.5: Execution time for different gadget sizes.

high). Both methods hide the control flow graph and allow the attacker only local view on small code blocks.

Resilience (strength against automated deobfuscators) is based on the runtime of a deobfuscator and the scope of the obfuscation transformation. The runtime grows *exponentially* with the size of the software and the branching level of the resulting graph, as a deobfuscator has to traverse through the entire graph to reconstruct the control flow. For example, splitting a small program (100 assembly instructions) into gadgets of 12 to 15 instructions and building a gadget graph where every node has 2 to 3 child nodes, yields to more than 1800 different paths through this graph. In Collberg's classification, the scope of our transformation is "*global*". The combination of both measures results in the resilience level "*strong*".

We furthermore used two state-of-the-art reverse engineering tools to evaluate the strength of the static part of our approach. At first, we tried to reconstruct the program's control flow with the disassembler IDA Pro 5.6. Table 3.1 compares the automated disassembling rates for the original versions of the code and the obfuscated ones. The values in the table are the percent of successfully reconstructed areas. While IDA Pro was able to reconstruct nearly 38% of the original AES code, the percentage for the obfuscated version declined to about 10%. For the MOV benchmark, the difference was even larger. The results show that for both the AES algorithm and the MOV benchmark, the obfuscated version was much more difficult to reconstruct for IDA Pro. The huge differences between the two examples was caused by different amount of obfuscated code. While for the MOV benchmark the entire code was obfuscated, in the AES example only the algorithm itself was obfuscated. IDA Pro was able to reconstruct non-obfuscated parts of the code correctly, but failed at reconstructing obfuscated code. The disassembler is not able to determine the jump targets of the branching function without actually executing it.

The second tool we used for evaluation is Jakstab [113] which aims at recovering control flow graphs. Jakstab was not able to resolve the indirect jump at the end of the branching function of our sample program. Although it successfully extracted some of the jump targets from the lookup table, the correct order of the jumps still remained unknown to Jakstab.

Although both tools implement methods for disassembling software and reconstructing control flow graphs, it is not surprising to see them fail at breaking our proposed obfuscation tech-

AES algorithm		MOV benchmark	
original	obfuscated	original	obfuscated
37.96%	10.27%	100%	0.13%

Table 3.1: Amount of successfully reconstructed code areas (IDA Pro).

nique as they are not tailored to our particular implementation. Hence, for a more realistic evaluation we also discuss on what a possible deobfuscator for our approach would look like.

One of the main strengths of our approach is that obfuscated software does not contain an explicit representation of the graph structure. It is hidden inside the lookup table, which only reveals the direct successor of a gadget within a single trace during runtime. If an attacker wants to manipulate the software (e.g. remove a copy protection mechanism) he could pursue the following two strategies:

- **Reconstructing the entire graph.** Without obfuscation, an attacker would search for the copy protection code inside the software and then remove it. In our diversified version of the software, however, multiple different versions of the copy protection are distributed over the entire code. Moreover, they are split into small blocks to fit into the gadgets. An attacker could execute every possible trace of the software and so reconstruct the entire control flow graph. The result would, without doubt, reveal the structure of the code as the individual traces can be analyzed separately. However, the enormous number of possible paths through the graph makes this approach time consuming.

- **Removing diversity of a single trace.** Alternatively, the attacker could remove the copy protection code from one trace and then make this trace valid for all inputs (i.e. remove diversity). The main challenge of this approach is, that the attacker has to analyze and understand the entire trace to be able to identify and remove modifications to operands that were inserted during obfuscation time to bind the code to a specific input.

Neither strategy can likely be performed without human interaction. In the first one, a large number of variants of the same copy protection mechanism would have to be identified and removed manually from the individual traces. In the second strategy, a human deobfuscator would have to analyze an entire trace to be able to identify the inserted modifications that make the trace specific to a single input. We believe, that this high amount of manual effort significantly raises the bar for reverse engineering attacks.

CHAPTER 4

Hardware-assisted Control Flow Obfuscation

Theoretical results by Barak et al. [12] indicate that provably secure code obfuscation in general is not possible. Thus, its resilience remains unclear and ultimately depends only on available resources and patience of the attacker. To tackle this dilemma we propose enhancing the strength of code obfuscation with hardware-assistance in the system's microprocessor. Unlike heavy, full-fledged hardware such as dedicated secure co-processors our concept requires only small modification to the hardware and could be easily implemented in future generations of microprocessors. Our theoretical framework towards lightweight software protection mechanisms in hardware can be considered as analogous to proposals in academia about a decade ago for the implementation of cryptographic primitives directly into the microprocessor through instruction set extensions [191, 192, 193]. Intel as well as other vendors of microprocessors embraced this concept in recent years and implemented AES instructions in their microprocessors in order to improve the speed of encryption and decryption tasks [93], making use cases such as full disk encryption much more efficient.

4.1 General Idea

There are many application scenarios in which it is desired to make the analysis of a software product as complicated and time consuming as possible. For example, think of the gaming industry. For the commercial success of computer games, the first weeks of sale are the most crucial ones. Traditionally, most people buy new games shortly after the release and weekly sales start decreasing quickly after this first rush. Thus, a copy protection mechanism that is able to prevent a generic break during the first weeks after release can significantly contribute to the commercial success of a game.

In this chapter, we present a novel software protection technique that combines two limiting factors of code analysis in order to delay a successful attack against an obfuscated program. The

first aim of our scheme is *strong hardware-software binding*. Program execution — and even more important — program analysis is made impossible without access to a particular hardware instance. It has been shown in the literature that by letting the correct execution of software be dependent on the presence of a particular hardware module, static code analysis can be prevented effectively [217]. By analyzing the software alone the attacker only gets an incomplete view on the program that lacks the functionality of the hardware. However, this argument entirely ignores the fact that an attacker who has access to the required hardware instance can still analyze the program dynamically. In academic research, dynamic analysis has often been accepted as an unpreventable type of analysis. The core argument is that if an attacker has access to the machine the software is running on, he can observe the program execute instruction by instruction in a debugger. While this is certainly a time consuming task, reality looks quite different. Dynamic reverse engineering efforts typically focus on a narrow scope of the analyzed program only, while the rest of the program is analyzed statically. In our approach we leverage this observation by applying an obfuscation technique that forces the attacker into *dynamic code analysis of the entire program* on one particular hardware instance — an extremely limited and time consuming analysis context. Thus, while code analysis cannot be prevented entirely, we aim at drastically restricting its power by forcing the attacker into the severe limitations of the analysis context.

In more details, our concept is based on the idea of modifying Intel's hardware based AES implementation (AES-NI) in order to make it suitable for code obfuscation scenarios. This allows us to implement code obfuscation schemes that make the execution of a program dependent on the existence of a cryptographic key in a protected memory location of the host system's microprocessor.

4.2 Scenario

In this section, an overview of our concept for hardware-assisted software obfuscation is given. Figure 4.1 explains its core components. The *hardware-software binding* is based on a cryptographic key, which is securely stored in the hardware and required for executing the software. This key is different for each instance of the software. The distribution of cryptographic keys that are used for code obfuscation purposes at runtime is a challenging task. A potential attacker has full access to the system on which the protected software is executed. Thus, in such a scenario, which is called a *white-box attack context*, key distribution has to be performed in a way that an attacker is not able to intercept the keys. Placing a program-instance specific cryptographic key into the hardware at manufacturing time is not practical. On the one hand, program developers have to know the hardware's specific key at compile time in order to be able to customize the software to make its execution depend on a specific key. On the other hand, the key has to be kept secret in order to be able to protect the obfuscation scheme. Depending on the application scenario, different ways of key distribution are possible.

In some cases, the software is bundled with the hardware. Think of set-top boxes or DSL modems which often come pre-configured by the service provider as part of the contract with the customer. The service provider is able to customize the software for a unique obfuscation key and install this key in the hardware. However, placing a program-instance specific obfuscation key into the hardware before delivering it to the end user is not practical in scenarios where

hardware and software are sold separately to the user. Neither it is possible to securely install a secret key after delivery of the hardware without revealing it to the user. Public key cryptography would solve the problem (a PKI could be used to securely place a symmetric obfuscation key in the processor). However, it is difficult to justify its costly adaption in a system that is focused on lightweight hardware and moderate modifications of existing processor architectures.

We solve the problem of key distribution in scenarios where hardware and software are sold separately with an approach that uses two types of symmetric keys — a processor-specific *hardware key* and a program-instance specific *obfuscation key*. The hardware key is placed into the user's hardware at manufacturing time and is used to protect the obfuscation key, which is created at obfuscation time.

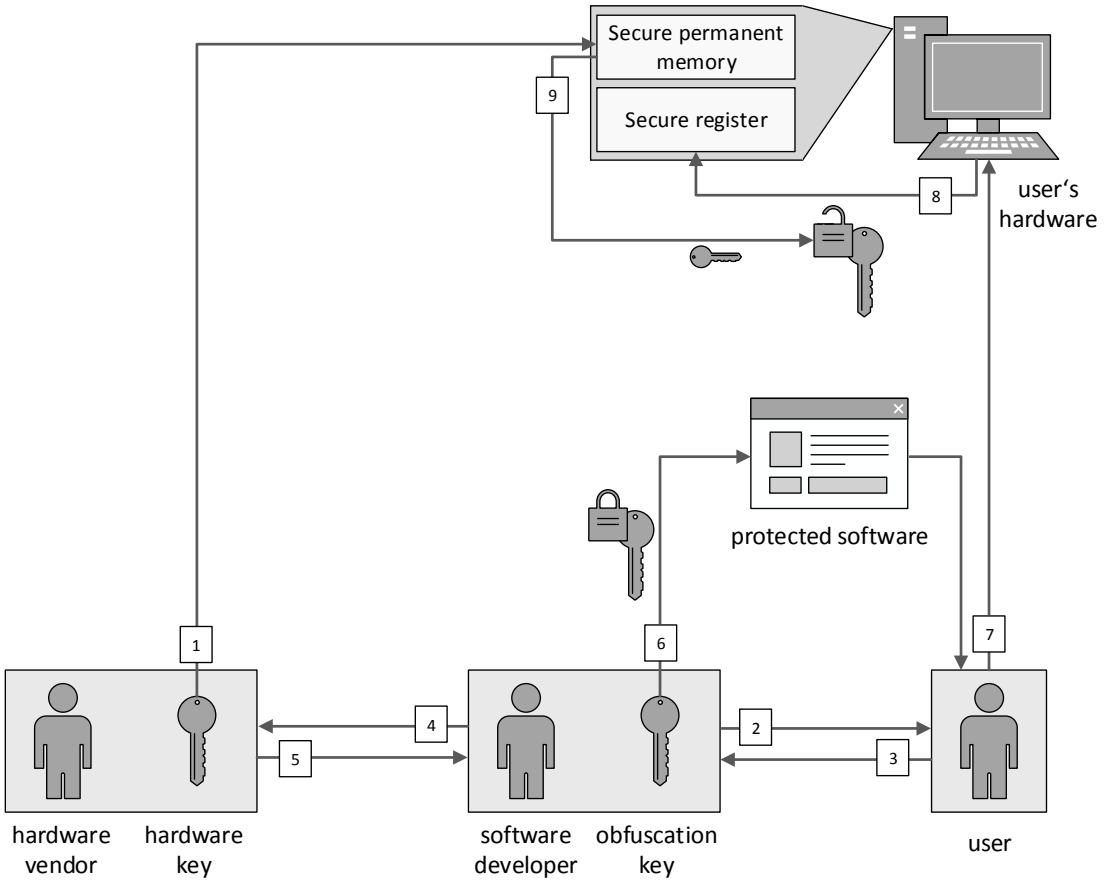

Figure 4.1: Key distribution in the proposed software protection approach.

In our concept both keys are stored in the user's hardware in special storage locations that are not accessible for the user. The *hardware key* is permanently stored in the user's hardware (marked with "1" in Figure 4.1). The actual storage concept for this key is not predefined by our approach. The key can either be written by the hardware vendor at manufacturing time

to a protected PROM located in the hardware's microprocessor or it can be derived from an intrinsic Physically Unclonable Function (PUF) [186]. In both cases, the hardware vendor keeps a copy of the key together with the processor ID. The software-specific *obfuscation key* (i.e. each protected program instance has a unique obfuscation key) is stored in a so called *secure register* at runtime of the program. We define a secure register as a special processor register, which is not accessible to the user. Section 4.3 explains the use of secure registers in detail.

The scenario for protected software distribution and execution is as follows: First, the software developer requests the unique processor ID of the user's system in order to be able to customize the software to the specific hardware instance (steps 2 and 3 of Figure 4.1). Then, the developer generates an obfuscation key, builds a software instance whose correct execution is dependent on this key and sends the key together with the processor ID to the hardware vendor (4). The vendor looks up the hardware key using the processor ID, encrypts the obfuscation key with the hardware key and sends the encrypted version back to the software developer (5). The software developer adds the encrypted obfuscation key to the customized version of his software and delivers it to the user (6). At runtime, when the user executes the protected program on his hardware (7), the hardware key is used to decrypt the obfuscation key (8), which is then stored in a secure register (9). At that point in program execution, the obfuscation key is used by the software to determine the correct execution path (see Section 4.4).

The latter two steps need to be performed secretly in the microprocessor. Even an attacker who has full access to the system should not able to intercept the obfuscation key during the decryption process. In the following section we explain our proposed AES-SEC processor instructions that make this secure distribution of keys possible.

4.3 AES-SEC: AES-NI in a White-box Analysis Context

In our approach cryptographic operations need to be performed in hardware. Since 2010 most microprocessors of Intel's x86 architecture include the "Advanced Encryption Standard New Instructions (AES-NI)" extension which provides six new instructions for AES encryption and decryption tasks as well as round key generation. None of these instructions implement the entire algorithm, instead it is split into small subtasks (e.g. the instruction `AESENC` is used for the calculation of one encryption round). Thus, an entire encryption/decryption cycle is performed by executing the instructions representing the subparts of the algorithm in the correct order. In Intel's reference implementation, the cryptographic key as well as intermediate states of the algorithm are stored in 128-bit wide registers (`xmm0` to `xmm15`) of the Streaming SIMD Extensions (SSE) [166].

While Intel's hardware implementation of AES is the prime candidate for cryptographic tasks in scenarios where encryption/decryption is performed on a trusted system (i.e. systems that are not under full control of an attacker), it is not possible to use it in a software protection context. An attacker who has full access to the system the algorithm is running on would be able to intercept the cryptographic keys and thus break the protection scheme. In the following, we introduce a set of modifications to the AES-NI instruction set that makes it suitable in application scenarios with a white-box analysis context (such as required for the code obfuscation scheme described in Section 4.4). We call this concept *AES-SEC*. The basic idea of AES-SEC is to turn

the hardware-based algorithm into a *virtual black-box* from which an attacker can only analyze input and output behavior.

The first and obvious limitation of the original AES-NI instruction set in a white-box analysis context is the storage location of the cryptographic key. Intel's x86 architecture does not provide storage locations that can be hidden from dynamic code analysis (e.g. observing the program execution in a debugger reveals the content of the processor's registers). To mitigate this problem, we propose the implementation of a new set of processor registers. These so-called *secure registers* are accessible only through a limited set of instructions which are derived from Intel's AES-NI instruction set extension. Even these newly introduced instructions can use the secure register only for calculating and storing intermediate results and do not provide functionality for moving sensitive data to unprotected storage locations. Thus, at no point in time, sensitive data from a secure register can leave the CPU and neither an attacker nor software which is executed on the system are able to access the secure registers in a way that enables them to reveal its content. While the implementation of an entirely new type of registers adds without doubt additional complexity to the microprocessor architecture, the general concept of secure registers would enable various security related use cases. For instance, the secure storage of cryptographic keys for full-disk encryption is a yet unsolved problem [20, 150] which would benefit immensely from the existence of secure registers. Thus, the proposed hardware modifications are not limited to obfuscation purposes only. In this thesis, however, we lay the focus on its application in software protection scenarios.

However, even in the presence of secure registers that cannot be read directly from the software, the fact that the algorithm is split up into separate instruction calls for each encryption round makes it easy for an attacker to calculate the content of the hidden key storage register by performing a reduced number of AES rounds. For example, an attacker could modify the program code to only execute one single decryption round and then perform a differential cryptanalysis in order to calculate the key. As a countermeasure we propose several small modifications to the AES-NI instruction set extension that allow the implementation of AES-NI based cryptography in scenarios with untrusted host systems (e.g. for code obfuscation purposes). Similar to the solution for protecting the cryptographic key, the first modification of the instructions is to store all intermediate states of the algorithm in secure registers instead of using SSE registers to prevent an attacker from directly accessing security relevant data. The second modification aims at preventing key extraction by running a reduced number of AES rounds. We propose the implementation of five new instructions that heavily build upon the existing AES instructions but extend their functionality to support application scenarios in untrusted environments. The basic idea of AES-SEC is the implementation of a *round counter* that tracks how many encryption rounds have been performed. The value of the counter is also stored in a secure register in order to prevent an attacker from manipulating it. At the beginning of a decryption cycle a new *initialization instruction* (`AESLOADVALSEC`) is executed, which copies the input data to a secure register and sets the round counter to 0. Then, each time the encryption instruction of AES-SEC is executed, it increments the counter. AES-NI has a separate instruction for the final round of AES. In our approach of AES-SEC this instruction additionally implements a check if the required number of rounds has been performed. It is further necessary to prevent that an attacker is able to control the input and output registers of the AES-SEC instructions. Thus, we

designed the AES-SEC instructions without operands for the input and output registers so that the instructions perform on fixed secure registers. As it only makes sense to use the proposed AES-SEC instructions and the secure registers in combination, the restriction to fixed registers does not reduce flexibility of the approach.

Our implementation uses 13 secure registers. The first register stores the AES key, either the hardware key or the obfuscation key. The second register stores the input data and intermediate states respectively, while the third register serves as a secure status register similar to the flags register that is present in most of today's microprocessor architectures. It stores the AES-SEC round counter as well as the state of the algorithm. Registers 4 to 13 store a copy of all 10 round keys. We further swap the encryption and decryption routines, which is possible because of the commutativity property of AES if only one block of input data is processed. Thus, we use the encryption instructions for decryption. With this modification, a secure version of the `AESIMC` instruction, which is required for generation of decryption round keys, is not necessary as only one direction of the AES algorithm is used at runtime of a protected program. Thus, round key generation can be initialized using a modified version of the `AESKEYGENASSIST` instruction, which is normally used for the generation of encryption round keys.

The entire concept of AES-SEC is built on the requirement that even an attacker, who can execute the new instructions in arbitrary number and order, must still not be able to move cryptographic keys or intermediate AES states from the secure registers to unprotected storage locations. Both the round counter and the status flags are used by the proposed instructions to verify that a particular action can be performed on the current state of the AES algorithm without revealing sensitive data to a public storage location. Figure 4.2 visualizes the virtual black-box paradigm of AES-SEC. Except from input and output values, the entire AES algorithm is performed in a virtual black-box that does not allow extraction of cryptographic keys or intermediate results by an attacker. An in-depth specification of AES-SEC and secure registers can be found in Appendix A. Section A.1 describes the proposed AES-SEC instructions and compares them to the original instructions from the AES-NI instruction set extension. The internals of the AES-NI instructions are explained in Section A.2. Section A.3 demonstrates one decryption round in detail. Note that the entire algorithm is performed on secure registers. Only the instruction for the final AES round can write the calculated output to a public register after a full cycle of AES rounds has been performed.

4.4 Control Flow Obfuscation with AES-SEC

In this section, we demonstrate how the proposed modifications of the AES-NI instruction set can be used for control flow obfuscation. The described technique implements a slightly modified version of a concept for control flow diversification which we introduced in Chapter 3. The first difference is that we perform obfuscation on a basic block layer and reduce the number of possible path signatures at each node of the control flow graph to one (i.e., the path signature at a specific node is independent from which path in the control flow graph is taken to the node). As the hardware-software binding already effectively prevents class breaks, we are able to leverage the security property of the control flow diversification. This allows us to significantly improve performance at obfuscation time. The second difference is that the calculation of jump targets is

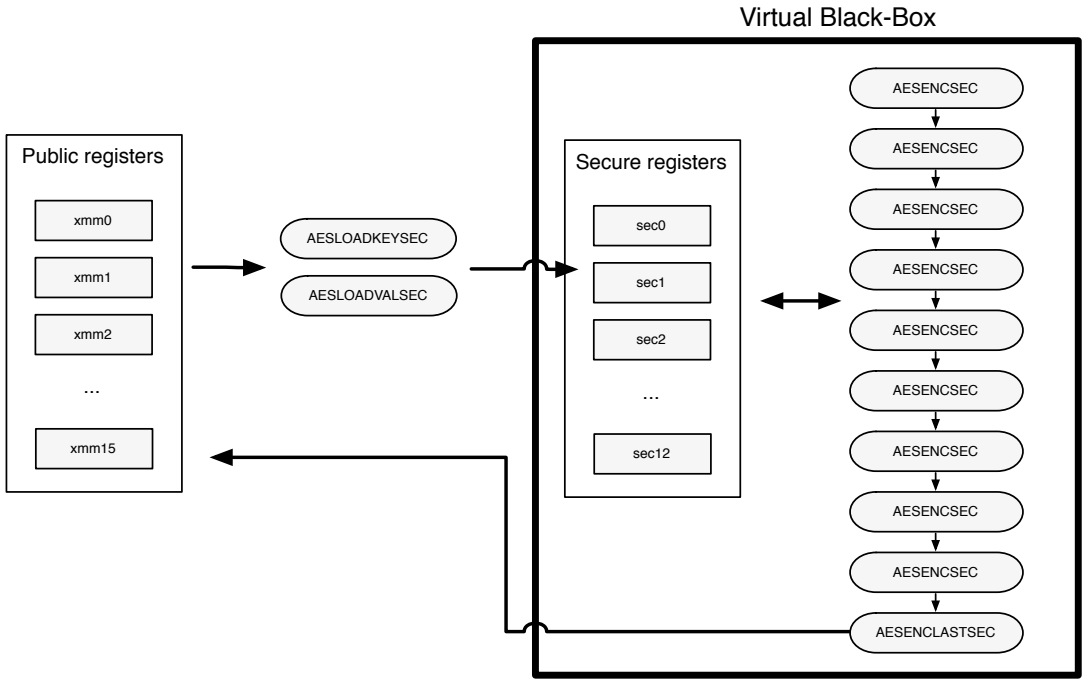

Figure 4.2: The virtual black-box paradigm of AES-SEC.

not entirely performed in software. The path signature at a particular basic block is the encrypted representation of the following jump target's address and the AES-SEC instructions are used to decrypt the address using the obfuscation key. The round keys derived from the obfuscation key are written to the secure registers during a special *key initialization process* which is part of the protected program and executed at launch time. As depicted in Figure 4.1, the protected program contains a representation of the obfuscation key which is encrypted with the hardware key. Using the AES-SEC instructions, the obfuscation key can be decrypted and stored in a secure register without revealing it to an attacker who has full access to the system during key initialization. In detail, the key initialization process first generates the round keys of the hardware key, then decrypts the obfuscation key and finally generates the round keys of the obfuscation key. In the following, the construction of an obfuscated control flow graph as well as the calculation of jump targets at runtime are explained in detail.

4.4.1 Obfuscation time

The path signatures of all basic blocks of the control flow graph are created at obfuscation time. Algorithm 4.1 shows the algorithm for this process in pseudocode. First, each basic block of the program is assigned with an encrypted representation of the following block's starting address. For this process, a program-instance specific obfuscation key is generated. Then, the algorithm iterates through the set of edges between blocks, where an edge in the control flow graph is the

connection between two blocks through an indirect jump. For each node the XOR between the encrypted starting addresses of the source and the target blocks is calculated. The result is stored as a part of the path signature inside the source block (*path value*).

At obfuscation time, the complexity for calculating the path signature of the entire control flow graph is in $O(n)$, where n represents the total number of edges, as every edge has to be visited only once to be able to create the path signature for the entire control flow graph and both nodes and edges of the control flow graph are known to the obfuscator.

4.4.2 Runtime

During runtime the path signature is constantly updated by XORing it with the path value of the current block. Thus, the signature is unknown to an attacker without knowing a valid path to the jump instruction he aims at resolving. To finally calculate the correct jump target of a block, the cumulated path signature is decrypted with the program's obfuscation key using the AES-SEC instructions. Note that an attacker is not able to intercept the obfuscation key during jump target decryption as the entire process is performed on secure registers only. Thus, without access to the hardware instance that contains the correct hardware key, an attacker is not able to calculate jump targets.

Even in case that the attacker as access to the required hardware instance, complexity of analysis is high. In contrast to the complexity at obfuscation time which is in $O(n)$, at runtime an attacker lacks knowledge about the control flow graph, thus is not able to calculate jump targets locally because of the path signature. Static control flow reconstruction approaches such as [115] will fail, because of the limited control flow information that is available through static analysis. In order to be able to break the hardware-software binding, the attacker has to perform a dynamic analysis over the entire program (i.e., analyzing the entire control flow graph) on one specific machine. While this is definitely possible, considerably more time and effort on the attacker side is necessary compared to previous obfuscation techniques.

Algorithm 4.1: Construction of the path signature in pseudocode

```
1 generate random key k;
2 forall the basic blocks do
3     encrypt jump target with k;
4 forall the edges between basic blocks do
5     calculate signature of source block by XORing enc. jump targets of source and target;
```

4.5 Evaluation

For evaluation purposes, we implemented a prototype using the existing AES-NI instruction set and SSE registers. Of course, such an implementation lacks essential security properties, thus rendering it applicable for performance evaluation purposes only. Still, it can demonstrate the practical utility of the approach.

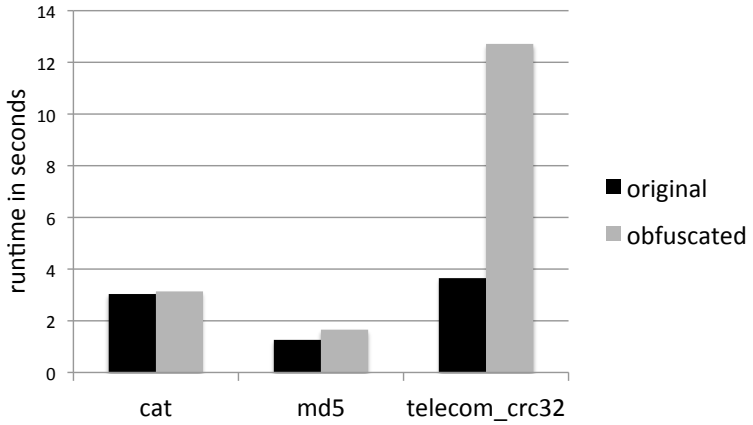

Figure 4.3: Performance evaluation.

Our prototype obfuscator is based on the LLVM compiler infrastructure [124]. Usually, so-called *passes* are used to optimize code during the compilation process. In contrast, we implemented an *obfuscation pass* which applies our software protection scheme to any program at compile time. We selected two simple Unix programs (`cat` and `md5`) as well as a test case from the benchmark suite cBench [84]. Figure 4.3 shows the results. Unsurprisingly, the differences in performance between the original programs and the obfuscated ones heavily depend on their structure. While the test case `telecom_crc32` is very memory intensive and evaluates the raw performance of the algorithm, both `md5` and `cat` are IO intensive. The results of the obfuscated version of `telecom_crc32` clearly show that the applied obfuscation significantly decreases the performance of the isolated algorithm. On the other hand, `md5` and `cat` demonstrate that in more IO intensive programs, the bottleneck from the obfuscation does not significantly contribute to the overall performance of the program.

4.5.1 Limitations

The most important limitation of our approach is its inability to run more than one obfuscated program at the same time. In x86, during context switches programs save their state (i.e. register values) on the stack. However, in our scenario it is not possible to store sensitive keys on the stack, where they can be accessed by an attacker. Thus, we have to ensure, that cryptographic keys and intermediate states do not leave the microprocessor at any time. Supporting multitasking of obfuscated programs would require considerably more modifications to the microprocessor design. Still, one obfuscated program that uses the secure registers can be run in multitasking mode together with an arbitrary number of traditional programs.

CHAPTER 5

Covert Computation – Hiding Code in Code

During the last decade, malware detection has become a multi-billion dollar business and an important area in academic research alike. One of the key findings of Chapter 2 is the strong focus on malware detection in current code analysis research. However, static analysis which is still the predominant technique for commercial 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 [42, 91]. 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, and 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. [43], aims at solving some of 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* [49] and *equivalent instruction replacement* [69]. 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 chapter we demonstrate that this fundamental prerequisite for semantic-aware malware detection is difficult to achieve. Our concept of COVERT COMPUTATION implements program functionality in side effects of the microprocessor that are not covered by a simple machine model. In contrast to packer-based obfuscation techniques which hide code in data sections that cannot be evaluated through static analysis, we hide (potentially 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 significantly more difficult to detect through semantic analysis than existing obfuscation techniques.

5.1 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 subtile 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 chapter, a comprehensive analysis of side effects in the x86 platform is conducted (see Table 5.1 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.

5.1.1 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.

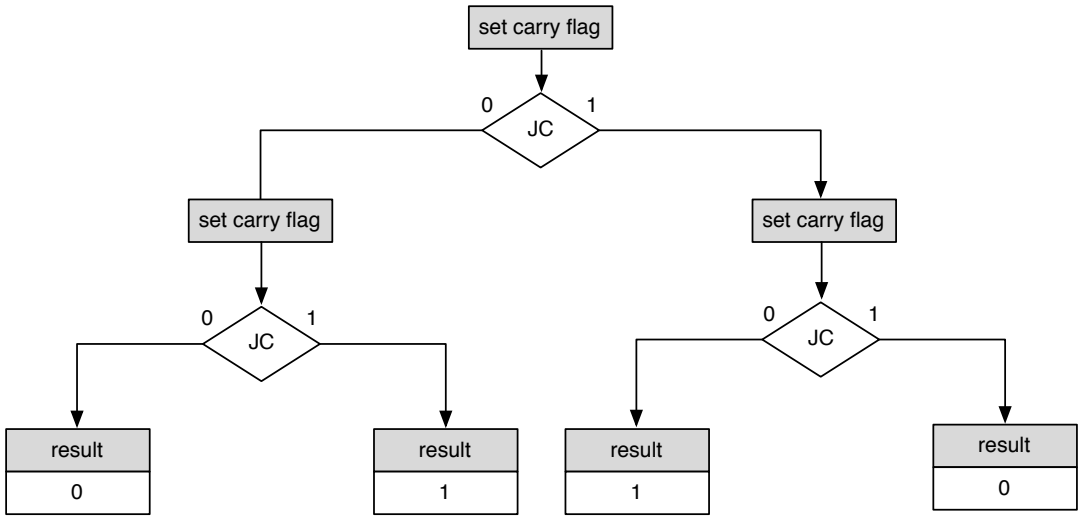

Figure 5.1: XOR using the carry flag.

```
;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 5.1: XOR using the carry flag.

Figure 5.1 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 5.1. 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 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.

5.1.2 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 5.2 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.

```
;Location of input value: EAX

TEST EAX, EAX
JNZ SHORT 70

    ⇓

MOV ECX, EAX
INC ECX
LOOP 70
```

Listing 5.2: Conditional jump with LOOP instruction.

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

Table 5.1: Side effects of the x86 architecture.

```
;Location of input value: EAX

SUB EAX,100

    ⇓

MOV ECX,100
XCHG EAX,ECX
LOOP -1

;Location of output value: EAX
```

Listing 5.3: SUB with LOOP instruction.

Listing 5.3 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 5.1 to 5.3 the value of ECX is modified. If the register is used at this location, its value has to be saved and restored.

5.1.3 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 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.
- 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 5.4 shows a code fragment which implements the functionality of the INC instruction using side effects of the LODS instruction.

```
;Location of input value: EAX

INC EAX

    ⇓

XCHG EAX, ESI
LODS
XCHG EAX, ESI

;Location of output value: EAX
```

Listing 5.4: Arithmetic operations with string instructions.

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 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.

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 5.5 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).

```
;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 5.5: Arithmetic operations (ADD) with string instructions.

Listing 5.6 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.

```
;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 5.6: Arithmetic operations (SUB) with string instructions.

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 5.4 to 5.6 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.

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

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

    ⇓

MOVS EDI, ESI
MOV EAX, ESI

;Location of output value: EAX
```

Listing 5.7: MOV with string instructions.

For example, MOVS increments or decrements both ESI and EDI and is otherwise equivalent to MOV [EDI],[ESI][1]. Listing 5.7 shows an example of replacing MOV instructions with string instructions.

5.1.4 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 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* [78]. 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

[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.

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 thesis; 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].

5.2 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 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 [135]. 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* [175, 181]) 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 [124] to insert the covert instructions directly into the hardware-specific assembly representation of the code. Figure 5.2 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 the obfus-

[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.

cating 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 to 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.

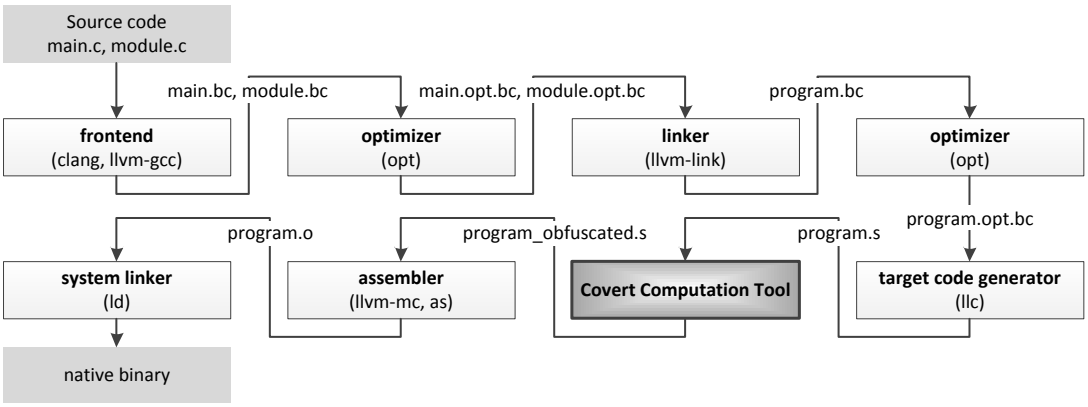

Figure 5.2: Intercepting the compilation process of LLVM.

5.3 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. [149], 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. [43].

5.3.1 Resilience against semantic-aware detection approaches

For semantic-aware malware detection [43], 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.

IR normalization: The approach introduced by Christodorescu et al. [43] 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.

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 [196], 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 fragments, i.e., instruction sequences that do not change the values of the watched variables. The following strategies were implemented by Christodorescu et al. [43]: (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.

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 Figure 5.3 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 Figure 5.3 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 << 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 ef-

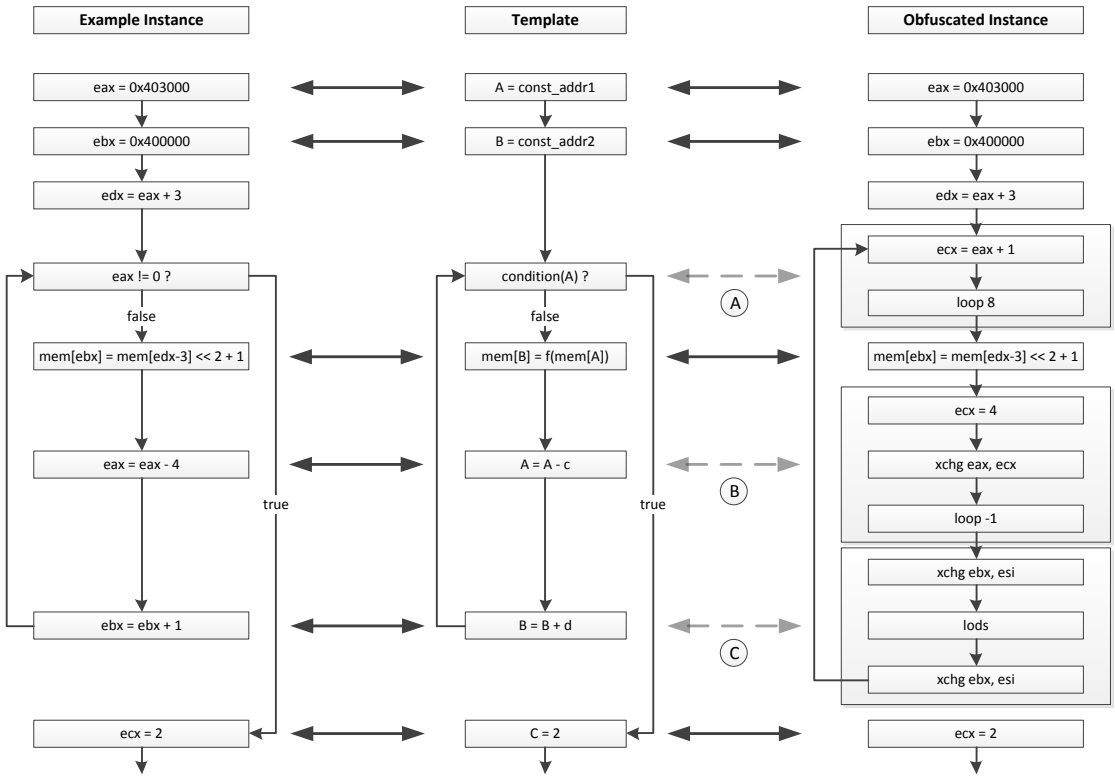

Figure 5.3: Resilience against semantic-aware malware detection.

fects. 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 [196], we know that deciding equivalence is not possible in general.

Another important aspect is diversity. Christodorescu et al. [43] 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.

5.4 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.

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

Table 5.2: Impact on binary size.

5.4.1 Prototype implementation

We measured performance and binary size overhead using our prototype implementation that intercepts the compilation process of LLVM [124]. 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.

Binary size. Binary size increase depends on the frequency of instructions that are replaced with semantically equivalent sequences of instructions. Table 5.2 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 [210], as acceptable.

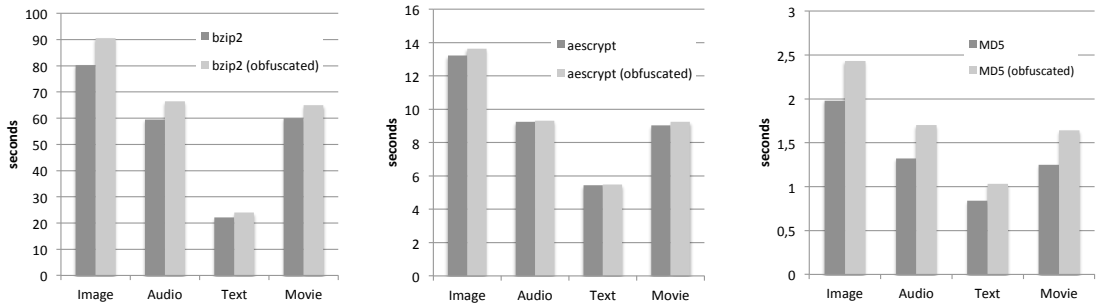

Figure 5.4: Performance analysis with aescrypt, MD5, and bzip2.

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,

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

	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

Table 5.3: Opcode frequency in selected malware samples (D=Dorkbot, G=Gamarue, S=Sality, Y=Yeltminky).

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 for each program (normal and obfuscated) and calculated the arithmetic average of ten independent runs each. The results can be found in Figure 5.4.

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 5.3) 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`.

5.4.2 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 [103]. 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 5.1. Figure 5.5 (a) 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 Figure 5.5 (b) 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 [218], 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.

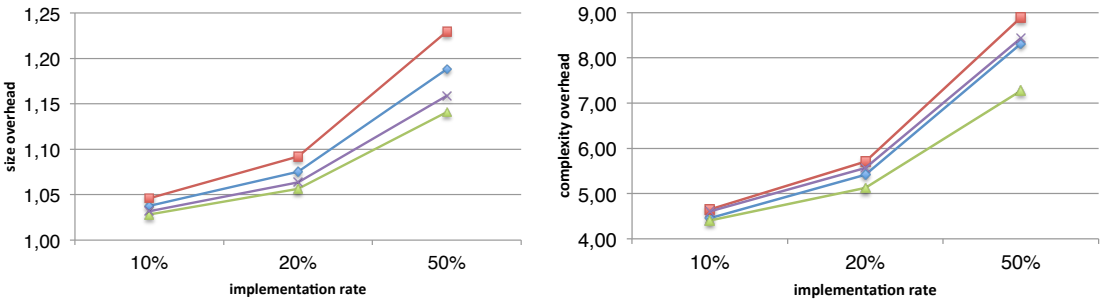

Figure 5.5: Theoretical evaluation of program (top) and complexity (bottom) overhead for four malware samples.

5.4.3 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 [16]. 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 5.3 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 [16], 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 [16]. 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 mitigation strategy, various host

code fragments with different opcodes could 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 [15]) 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.

CHAPTER 6

Conclusions

6.1 Definition of Attack Scenarios and Classification of Obfuscation Techniques

We analyzed the fundamental question to what extent code obfuscation techniques proposed during the last 25 years still provide reasonable protection of programs against state-of-the-art code analysis techniques and tools. The evaluation of different classes of obfuscation in specific attack scenarios showed that the answer heavily depends on the goals of the attacker and his available resources.

While — at least in theory — completeness of code analysis seems possible (and most of the analysis approaches introduced in academia indeed work for small and specific examples), real-world programs of large size can be considered significantly harder to analyze. A major limiting factor for code analysis is that the high complexity of analysis problems often exceeds resource constrains available to the analyst, thus making it fail for complex programs. Therefore, very simple obfuscation techniques can still be quite effective against pattern matching or static analysis. This explains the unbroken popularity of code obfuscation among malware writers. However, dynamic analysis methods, in particular if assisted by a human analyst, are much harder to cope with, which makes code obfuscation for the purpose of intellectual property protection a highly challenging task.

Another crucial observation is the strong focus on malware detection in today's code analysis research. The majority of recent literature provides methods for the classification of the maliciousness of programs purely based on the identification of obfuscation techniques that are typically used in malware. However, the analysis of the actual functionality of a program is out of scope of these works. Substantially fewer research has been done in academia for analyzing general (non-malware) programs and tool support is worse. As a consequence, today, analysis of this class of programs is mostly done via manual inspection.

The arms race between code obfuscation and analysis is still ongoing and the fundamental challenge of devising software protection mechanisms that are resistant against a human analyst

remains. With today's code obfuscation techniques one has to assume that a dedicated attacker, who is willing to spend enough time and effort, will always be able to successfully analyze a program. Nevertheless, code obfuscation may work in weaker attack scenarios.

6.2 Development of a parameterizable control flow graph diversification scheme

In Chapter 3 we proposed a novel software obfuscation method, based on control flow diversification, which makes it difficult for an attacker to relate structural information obtained by running a program several times and logging its trace. By splitting code into small portions (gadgets) before diversification, we achieve a complex control flow graph and static analysis can only reveal very limited local information of the program. We practically evaluated the strength of our approach against automated deobfuscators and showed that it can dramatically increase the effort for an attacker. A performance evaluation showed observable slowdowns for very small gadgets sizes, due to the vast amount of inserted jumps. Versions with bigger gadgets, however, yield to very reasonable performance results.

6.3 AES-SEC: Modification of AES-NI instructions for application in a white-box analysis context

Furthermore, we introduced a novel software protection technique that combines strong hardware-software binding with an attack context restriction to pure dynamic analysis — two major limiting factors of reverse engineering. We proposed modifications to Intel's AES-NI instruction set in order to make it suitable for application in software protection scenarios and combined it with our control flow graph obfuscation scheme (Chapter 3). The AES-SEC instruction create a virtual black-box that is able to perform a full AES encryption (decryption) round without leaking cryptographic keys or intermediate results of the algorithm to an attacker that has full control over the system the program is running on. We further introduced a key distribution system that allows a software developer to put a secret obfuscation key into the customer's processor without revealing it to the customer.

6.4 Covert Computation – Hiding Code in Code

Finally, 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 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.

APPENDIX A

Implementation Details of AES-SEC

A.1 Detailed specification of the proposed instructions

In the following, the new instructions of AES-SEC are described in detail:

AESLOADKEYSEC
The `AESLOADKEYSEC` instruction is used to load the hardware key into the secure register `sec0`. It further initializes the status flags in `sec2` to `01` and the counter for generated round keys to 0 (bits 2 to 5 of the secure status register). Listing A.1 shows the `AESLOADKEYSEC` instruction in pseudocode.

AESLOADVALSEC
The `AESLOADVALSEC` instruction is used to initialize the AES-SEC decryption cycle. It has one operand which sets the input data register. Further, the instruction sets the counter for performed AES rounds to 0 (bits 2 to 5 of the secure status register). Listing A.2 shows the `AESINITSEC` instruction in pseudocode.

AESENCSEC
The `AESENCSEC` instruction is based on the `AESENC` instruction from the AES-NI instruction set. The only two differences are that the instruction reads from and writes to secure registers instead of registers from the SSE instructions set extension and that the AES round counter is incremented by 1 each time the instruction is executed. Listing A.3 shows the `AESENCSEC` instruction in pseudocode.

storage location	stored data
sec0	AES key
sec1	input data / intermediate AES states
sec2	AES round counter (4 bits) and status flags (2 bits)
sec3 - sec12	round keys
secprom	hardware key

Table A.1: Secure registers.

AESENCLASTSEC
The AESENCLASTSEC instruction is based on AESENCLAST with the difference that it uses secure registers. The result of the last AES round can be written to either a public register such as xmm1 or to the secure register sec0 only if the counter for AES rounds reached the value 9. Hence, only if a complete AES cycle (AES-128 with 10 rounds) is performed it is accessible outside of the secure registers. Listing A.4 shows the AESENCLASTSEC instruction in pseudocode.

AESKEYGENASSISTSEC
AESKEYGENASSISTSEC is based on the AESKEYGENASSIST instruction. Similar to the AESENCSEC instruction, AESKEYGENASSISTSEC increments a counter in sec2 by 1 each time it is executed. While AESKEYGENASSIST has an operand for specification of an offset value, AESKEYGENASSISTSEC uses the counter from sec2 to calculate this offset. Listing A.5 shows the AESKEYGENASSISTSEC instruction in pseudocode.

A.2 Internals of the proposed instructions

```
AESLOADKEYSEC

sec0 <-- secprom
sec2 <-- 0x1
```

Listing A.1: AESLOADKEYSEC instruction.

```
AESLOADVALSEC xmm1

sec1 <-- xmm1
sec2 <-- sec2 & 0x3
```

Listing A.2: AESLOADVALSEC instruction.

```
AESENCSEC

sec1 <-- SubBytes(sec1)
sec1 <-- ShiftRows(sec1)
sec1 <-- MixColumns(sec1)
switch(sec2[2-5])
  case 0x0:
    sec1 <-- sec1 ⊕ sec3
  case 0x1:
    sec1 <-- sec1 ⊕ sec4
  case 0x2:
    sec1 <-- sec1 ⊕ sec5
  case 0x3:
    sec1 <-- sec1 ⊕ sec6
  case 0x4:
    sec1 <-- sec1 ⊕ sec7
  case 0x5:
    sec1 <-- sec1 ⊕ sec8
  case 0x6:
    sec1 <-- sec1 ⊕ sec9
  case 0x7:
    sec1 <-- sec1 ⊕ sec10
  case 0x8:
    sec1 <-- sec1 ⊕ sec11
sec2 <-- sec2 + 0x4
```

Listing A.3: AESENCSEC instruction.

```
AESENCLASTSEC

sec1 <-- SubBytes(sec1)
sec1 <-- ShiftRows(sec1)
if sec2 == 0x24
  xmm1 <-- sec1 ⊕ sec12
else if sec2 == 0x25
  sec0 <-- sec1 ⊕ sec12
```

Listing A.4: AESENCLASTSEC instruction.

```
AESKEYGENASSISTSEC

sec1 <-- [sec0[3]|sec0[2]|sec0[1]|sec0[0]]
sec2 <-- sec2 | 0x2
switch(sec2[2-5])
  case 0x0:
    sec3 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x1 |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x1:
    sec4 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x2 |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x2:
    sec5 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x4 |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x3:
    sec6 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x8 |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x4:
    sec7 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x10 |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x5:
    sec8 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x20 |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x6:
    sec9 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x40 |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x7:
    sec10 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x80 |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x8:
    sec11 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x1b |SubByte(sec1[1])]
    sec2 <-- sec2 + 0x4
  case 0x9:
    sec12 <-- [SubByte(RotByte(sec1[3])) ⊕ sec2 |SubByte(sec1[3])
        |SubByte(RotByte(sec1[1])) ⊕ 0x36 |SubByte(sec1[1])]
    sec2 <-- sec2 & 0x1
```

Listing A.5: AESKEYGENASSISTSEC instruction.

A.3 Full decryption cycle in AES-SEC

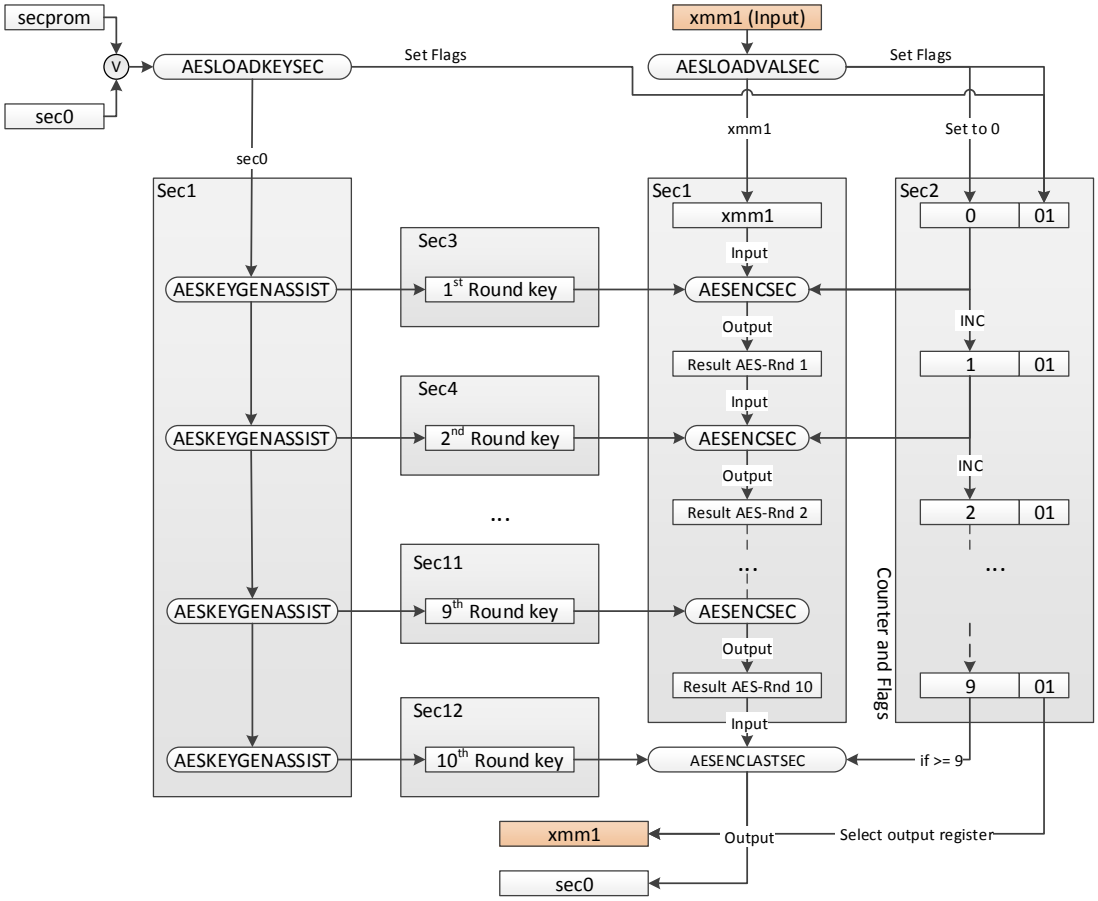

Figure A.1: Full decryption round using AES-SEC.

Bibliography

[1] T. Alves and D. Felton. TrustZone: Integrated Hardware and Software Security. *ARM white paper*, 3(4), 2004.

[2] B. Anckaert. *Diversity for software protection*. PhD thesis, Ghent University, 2008.

[3] B. Anckaert, B. De Sutter, and K. De Bosschere. Software Piracy Prevention through Diversity. In *Proceedings of the 4th Acm Workshop on Digital Rights Management*, pages 63–71. ACM, 2004.

[4] B. Anckaert, M. Jakubowski, and R. Venkatesan. Proteus: Virtualization for Diversified Tamper-Resistance. In *Proceedings of the ACM Workshop on Digital Rights Management*, pages 47–58. ACM, 2006.

[5] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program Obfuscation: A Quantitative Approach. In *Proceedings of the 2007 ACM Workshop on Quality of Protection*, pages 15–20. ACM, 2007.

[6] B. Anckaert, M. H. Jakubowski, R. Venkatesan, and C. W. Saw. Runtime Protection Via Dataflow Flattening. In *3rd International Conference on Emerging Security Information, Systems and Technologies (SECURWARE '09)*, pages 242–248. IEEE, 2009.

[7] G. Avoine, P. Junod, and P. Oechslin. *Computer System Security: Basic Concepts and Solved Exercises*. EPFL Press, 2007.

[8] A.M. Azab, P. Ning, and X. Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388. ACM, 2011.

[9] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[10] G. Balakrishnan and T. W. Reps. Analyzing Memory Accesses in x86 Executables. In Evelyn Duesterwald, editor, *Compiler Construction*, volume 2985, pages 5–23. Springer, 2004.

[11] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software Via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[12] 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.

[13] S. Bardin, P. Herrmann, and F. Védrine. Refinement-Based CFG Reconstruction from Unstructured Programs. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '11)*, pages 54–69, 2011.

[14] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[15] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. 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*, pages 8–8. USENIX Association, 2009.

[16] D. Bilar. Opcodes as Predictor for Malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[17] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a White Box AES Implementation. In *Proceedings of the 11th International Conference on Selected Areas in Cryptography*, pages 227–240. Springer, 2005.

[18] P. Biondi and F. Desclaux. Silver Needle in the Skype. *Black Hat Europe*, 6:25–47, 2006.

[19] N. Bitansky, R. Canetti, S. Goldwasser, S. Halevi, Y.T. Kalai, and G.N. Rothblum. Program Obfuscation with Leaky Hardware. In *Advances in Cryptology–Asiacrypt 2011*, volume 7073, pages 722–739. Springer, 2011.

[20] E.-O. Blass and W. Robertson. TRESOR-HUNT: attacking CPU-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 71–78. ACM, 2012.

[21] R.R. Branco, G.N. Barbosa, and P.D. Neto. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Blackhat 2012*, 2012.

[22] J. Bringer, H. Chabanne, and E. Dottax. White Box Cryptography: Another Attempt. *IACR Cryptology Eprint Archive*, 2006:468, 2006.

[23] T. Brosch and M. Morgenstern. Runtime Packers: The Hidden Problem. *Black Hat USA*, 2006.

[24] D. Brumley, I. Jager, T. Avgerinos, and E.J. Schwartz. BAP: A Binary Analysis Platform. In *23th International Conference on Computer Aided Verification (CAV '11)*, pages 463–469, 2011.

[25] D. Bruschi, L. Martignoni, and M. Monga. Detecting Self-Mutating Malware Using Control-Flow Graph Matching. *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 129–143, 2006.

[26] J. Caballero, N.M. Johnson, S. McCamant, and D. Song. Binary Code Extraction and Interface Identification for Security Applications. In *Proceedings of Network and Distributed System Security Symposium (NDSS '09)*, 2010.

[27] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[28] J. Calvet, J. M Fernandez, and J.-Y. Marion. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 169–182. ACM, 2012.

[29] R. Canetti and R. Dakdouk. Obfuscating Point Functions with Multibit Output. *Advances in Cryptology–Eurocrypt 2008*, pages 489–508, 2008.

[30] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca. Software Salvaging Based on Conditions. In *Proceedings of the International Conference on Software Maintenance (ICSM '94)*, pages 424–433. IEEE, 1994.

[31] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned Program Slicing. *Information and Software Technology*, 40(11):595–607, 1998.

[32] J. Cappaert and B. Preneel. A General Model for Hiding Control Flow. In *Proceedings of the 10th Annual ACM Workshop on Digital Rights Management*, pages 35–42. ACM, 2010.

[33] J. Cappaert, N. Kisserli, D. Schellekens, and B. Preneel. Self-Encrypting Code to Protect against Analysis and Tampering. In *1st Benelux Workshop on Information and System Security*, 2006.

[34] R.S. Chakraborty, S. Narasimhan, and S. Bhunia. Embedded software security through key-based control flow obfuscation. In *Security Aspects in Information Technology*, pages 30–44. Springer, 2011.

[35] H. Chang and M.J. Atallah. Protecting Software Code by Guards. In *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 160–175. Springer, 2002.

[36] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6), 2002. ISSN 0098-5589.

[37] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 265–278, 2011.

[38] M.R. Chouchane and A. Lakhotia. Using Engine Signature to Detect Metamorphic Malware. In *Proceedings of the 4th ACM Workshop on Recurring Malcode*, pages 73–78. ACM, 2006.

[39] S. Chow, Y. Gu, H. Johnson, and V.A. Zakharov. An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In *Information Security*, pages 144–155. Springer, 2001.

[40] S. Chow, P. Eisen, H. Johnson, and P. Van Oorschot. White-Box Cryptography and an AES Implementation. In *Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2003.

[41] S. Chow, P. Eisen, H. Johnson, and P.C. Van Oorschot. A White-Box DES Implementation for DRM Applications. In *Digital Rights Management*, volume 2696, pages 1–15. Springer, 2003.

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

[43] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-Aware Malware Detection. In *26th IEEE Symposium on Security and Privacy*, pages 32–46. IEEE, 2005.

[44] M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith. Software Transformations to Improve Malware Detection. *Journal in Computer Virology*, 3(4):253–265, 2007.

[45] C. Cifuentes and K.J. Gough. Decompilation of Binary Programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[46] A. Cimitile, A. De Lucia, and M. Munro. A Specification Driven Slicing Process for Identifying Reusable Functions. *Journal of Software Maintenance: Research and Practice*, 8 (3):145–178, 1996.

[47] F.B. Cohen. Operating System Protection through Program Evolution. *Computers & Security*, 12(6):565–584, 1993.

[48] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[49] 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.

[50] C. Collberg, C. Thomborson, and D. Low. Breaking Abstractions and Unstructuring Data Structures. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 28–38. IEEE, 1998.

[51] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196. ACM, 1998.

[52] P.M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *30th IEEE Symposium on Security and Privacy*, pages 61–76. IEEE, 2010.

[53] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic Static Unpacking of Malware Binaries. In *16th Working Conference on Reverse Engineering (WCRE '09)*, pages 167–176. IEEE, 2009.

[54] K. Coogan, G. Lu, and S. Debray. Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 275–284. ACM, 2011.

[55] A. Cooper and A. Martin. Towards an open, trusted Digital Rights Management Platform. In *Proceedings of the ACM workshop on Digital Rights Management*, pages 79–88. ACM, 2006. ISBN 159593555X.

[56] A. Cozzie, F. Stratton, H. Xue, and S.T. King. Digging for Data Structures. In *Symposium on Operating Systems Design and Implementation (OSDI '08)*, 2008.

[57] J.R. Crandall, Z. Su, S.F. Wu, and F.T. 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.

[58] J.R. Crandall, G. Wassermann, D.A.S. de Oliveira, Z. Su, S.F. Wu, and F.T. Chong. Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines. *ACM SIGPLAN Notices*, 41(11):25–36, 2006.

[59] M. Dalla Preda and R. Giacobazzi. Semantic-Based Code Obfuscation by Abstract Interpretation. In *Automata, Languages and Programming*, pages 1325–1336. Springer, 2005.

[60] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque Predicates Detection by Abstract Interpretation. *Algebraic Methodology and Software Technology*, pages 81–95, 2006.

[61] M. Dalla 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.

[62] M. Dalla 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.

[63] M. Dalla Preda, R. Giacobazzi, S. Debray, K. Coogan, and G. Townsend. Modelling Metamorphism by Abstract Interpretation. *Static Analysis*, pages 218–235, 2011.

[64] S. Danicic, A. De Lucia, and M. Harman. Building Executable Union Slices Using Conditioned Slicing. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 89–97. IEEE, 2004.

[65] S. Danicic, M. Daoudi, C. Fox, M. Harman, R.M. Hierons, J.R. Howroyd, L. Ourabya, and M. Ward. Consus: A Light-Weight Program Conditioner. *Journal of Systems and Software*, 77(3):241–262, 2005.

[66] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[67] L. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. XIFER: A Software Diversity Tool against Code-Reuse Attacks. In *4th ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3 2012)*, 2012.

[68] Y. De Mulder, B. Wyseur, and B. Preneel. Cryptanalysis of a Perturbated White-Box AES Implementation. In *Progress in Cryptology - INDOCRYPT 2010*, pages 292–310. Springer, 2010.

[69] 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*, pages 152–165, 2009.

[70] S. Debray and J. Patel. Reverse Engineering Self-Modifying Code: Unpacker Extraction. In *17th Working Conference on Reverse Engineering (WCRE '10)*, pages 131–140. IEEE, 2010.

[71] N. Dedić, M. Jakubowski, and R. Venkatesan. A Graph Game Model for Software Tamper Protection. In *Proceedings of the 9th International Conference on Information Hiding*, pages 80–95. Springer-Verlag, 2007.

[72] J.C. Deprez and A. Lakhotia. A Formalism to Automate Mapping from Program Features to Code. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 69–78. IEEE, 2000.

[73] C. Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2008.

[74] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Computing Surveys*, 44(2):6, 2012.

[75] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.

[76] M.V. Emmerik and T. Waddington. Using a Decompiler for Real-World Source Recovery. In *In Proceedings of the 11th Working Conference on Reverse Engineering*, pages 27–36. IEEE, 2004.

[77] J. Ferguson and D. Kaminsky. *Reverse Engineering Code with IDA Pro*. Syngress, 2008.

[78] P. Ferrie. This sig doesn't run. *Virus Bulletin*, January 2012.

[79] J. Field, G. Ramalingam, and F. Tip. Parametric Program Slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 379–392. ACM, 1995.

[80] C. Fox, S. Danicic, M. Harman, and R.M. Hierons. ConSIT: A Fully Automated Conditioned Program Slicer. *Software: Practice and Experience*, 34(1):15–46, 2004.

[81] M. Franz. E Unibus Pluram: Massive-Scale Software Diversity as a Defense Mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, pages 7–16. ACM, 2010.

[82] B. Fu, S. Aravalli, and J. Abraham. Software Protection by Hardware and Obfuscation. In *Proceedings of the 2007 International Conference on Security Management (SAM 2007)*, pages 367–373, 2007.

[83] K. Fukushima, S. Kiyomoto, and T. Tanaka. Obfuscation mechanism in conjunction with tamper-proof module. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 665–670. IEEE, 2009.

[84] G. Fursin. Collective benchmark (cbench). 2010. [Online; retrieved April 28th, 2014], `http://ctuning.org/cbench/`.

[85] R. Giacobazzi. Hiding Information in Completeness Holes: New Perspectives in Code Obfuscation and Watermarking. In *6th IEEE International Conference on Software Engineering and Formal Methods (SEFM '08)*, pages 7–18. IEEE, 2008.

[86] R. Giacobazzi and I. Mastroeni. Making Abstract Interpretation Incomplete: Modeling the Potency of Obfuscation. In *Proceedings of the 19th International Symposium Static Analysis (SAS 2012)*, pages 129–145. Springer, 2012.

[87] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, 2005.

[88] P. Godefroid, M.Y. Levin, and D.A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of Network and Distributed System Security Symposium (NDSS '08)*, 2008.

[89] S. Goldwasser and G.N. Rothblum. On Best-Possible Obfuscation. In *Theory of Cryptography*, volume 4392, pages 194–213. Springer, 2007.

[90] L. Goubin, J.M. Masereel, and M. Quisquater. Cryptanalysis of White Box DES Implementations. In *Selected Areas in Cryptography*, volume 4876, pages 278–295. Springer, 2007.

[91] K. Griffin, S. Schneider, X. Hu, and T. Chiueh. Automatic Generation of String Signatures for Malware Detection. In *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2009.

[92] F. Gröbert, C. Willems, and T. Holz. Automated Identification of Cryptographic Primitives in Binary Programs. In *Recent Advances in Intrusion Detection*, volume 6961 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2011.

[93] S. Gueron. Intel advanced encryption standard (AES) instructions set. *Intel White Paper, Rev*, 3, 2010.

[94] S. Gueron. White box aes using intel's new aes instructions. In *Information Technology: New Generations (ITNG), 2013 Tenth International Conference on*, pages 417–421. IEEE, 2013.

[95] Y. Guillot and A. Gazet. Automatic Binary Deobfuscation. *Journal in Computer Virology*, 6(3):261–276, 2010.

[96] M.H. Halstead. *Elements of software science*. Elsevier New York, 1977. ISBN 0444002057.

[97] L.C. Harris and B.P. Miller. Practical Analysis of Stripped Binary Code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005.

[98] W.A. Harrison and K.I. Magel. A complexity measure based on nesting level. *ACM SIGPLAN Notices*, 16(3), 1981. ISSN 0362-1340.

[99] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981. ISSN 0098-5589.

[100] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Model Checking Software (SPIN)*, volume 2648, pages 235–239. Springer, 2003.

[101] B. Horne, L. Matheson, C. Sheehan, and R.E. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. In *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 141–159. Springer, 2002.

[102] S. Horwitz. Precise Flow-Insensitive May-Alias Analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, 1997.

[103] iSeclab. Anubis. 2009. [Online; retrieved April 28th, 2014], `http://anubis.iseclab.org`.

[104] G. Jacob, P.M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A Static, Packer-Agnostic Filter to Detect Similar Malware Samples. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 102–122. Springer-Verlag, 2012.

[105] M. Jacob, D. Boneh, and E. Felten. Attacking an Obfuscated Cipher by Injecting Faults. *Digital Rights Management*, pages 16–31, 2003.

[106] M. Jacob, M.H. Jakubowski, and R. Venkatesan. Towards Integral Binary Execution: Implementing Oblivious Hashing Using Overlapped Instruction Encodings. In *Proceedings of the 9th Workshop on Multimedia & Security*, pages 129–140. ACM, 2007.

[107] M. Jakubowski, P. Naldurg, V. Patankar, and R. Venkatesan. Software Integrity Checking Expressions (ICEs) for Robust Tamper Detection. In *Information Hiding*, volume 4567, pages 96–111. Springer, 2007.

[108] M.G. Kang, P. Poosankam, and H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, pages 46–53. ACM, 2007.

[109] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting Self-Modification Mechanism for Program Protection. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, pages 170–179. IEEE, 2003.

[110] A. Karnik, S. Goswami, and R. Guha. Detecting Obfuscated Viruses Using Cosine Similarity Analysis. In *1st Asia International Conference on Modelling & Simulation (AMS '07)*, pages 165–170. IEEE, 2007.

[111] D. Kholia and P. Wegrzyn. Looking inside the (Drop)Box. In *7th Usenix Workshop on Offensive Technologies (Woot '13)*, 2013.

[112] J. Kinder. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *Proceedings of the 19th Working Conf. Reverse Engineering (WCRE 2012)*, pages 61–70. IEEE, 2012.

[113] J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *20th International Conference on Computer Aided Verification (CAV '08)*, pages 423–427. Springer, 2008.

[114] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting Malicious Code by Model Checking. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 3548, pages 174–187. Springer, 2005.

[115] J. Kinder, F. Zuleger, and H. Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, pages 214–228. Springer, 2009.

[116] S.T. King and P.M. Chen. SubVirt: Implementing Malware with Virtual Machines. In *27th IEEE Symposium on Security and Privacy*, pages 14–pp. IEEE, 2006.

[117] C. Kolbitsch, P.M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X.F. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, pages 351–366. USENIX Association, 2009.

[118] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *30th IEEE Symposium on Security and Privacy*, pages 29–44. IEEE, 2010.

[119] C. Kolbitsch, E. Kirda, and C. Kruegel. The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 285–296. ACM, 2011.

[120] C. Krügel, W.K. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Usenix Security Symposium*, pages 255–270, 2004.

[121] F. Lanubile and G. Visaggio. Extracting Reusable Functions by Flow Graph Based Program Slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, 1997.

[122] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: Using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 399–412. ACM, 2010.

[123] T. László and Á. Kiss. Obfuscating C++ Programs Via Control Flow Flattening. *Annales Universitatis Scientarum Budapestinensis De Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.

[124] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[125] F. Leder, P. Martini, and A. Wichmann. Finding and Extracting Crypto Routines from Malware. In *IEEE 28th International Performance Computing and Communications Conference (IPCCC '09)*, pages 394–401. IEEE, 2009.

[126] J. Li, M. Xu, N. Zheng, and J. Xu. Malware Obfuscation Detection Via Maximal Patterns. In *3rd International Symposium on Intelligent Information Technology Application (LITA '09)*, volume 2, pages 324–328. IEEE, 2009.

[127] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *17th Network and Distributed System Security Symposium*, 2010.

[128] H.E. Link and W.D. Neumann. Clarifying Obfuscation: Improving the Security of White-Box DES. In *International Conference on Information Technology: Coding and Computing (ITCC '05)*, volume 1, pages 679–684. IEEE, 2005.

[129] H.E. Link, R.C. Schroeppel, W.D. Neumann, P.L. Campbell, C.L. Beaver, L.G. Pierson, and W.E. Anderson. Securing Mobile Code. Technical report, Sandia National Laboratories, 2004.

[130] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 290–299. ACM, 2003.

[131] B. Lomb and T. Guneysu. Decrypting HDCP-Protected Video Streams Using Reconfigurable Hardware. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig '11)*, pages 249–254. IEEE, 2011.

[132] R. Lyda and J. Hamrock. Using Entropy Analysis to Find Encrypted and Packed Malware. *Security & Privacy, IEEE*, 5(2):40–45, 2007.

[133] B. Lynn, M. Prabhakaran, and A. Sahai. Positive Results and Techniques for Obfuscation. In *Advances in Cryptology–Eurocrypt 2004*, pages 20–39. Springer, 2004.

[134] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid Static-Dynamic Attacks against Software Protection Mechanisms. In *Proceedings of the 5th Acm Workshop on Digital Rights Management*, pages 75–82. ACM, 2005.

[135] M. Madou, B. Anckaert, B. De Bus, K. De Bosschere, J. Cappaert, and B. Preneel. On the Effectiveness of Source Code Transformations for Binary Obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*, pages 527–533, 2006.

[136] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software Protection through Dynamic Code Mutation. In *Information Security Applications*, pages 194–206. Springer, 2006.

[137] M. Madou, L. Van Put, and K. De Bosschere. LOCO: An Interactive Code (De) Obfuscation Tool. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 140–144. ACM, 2006.

[138] M. Madou, L. Van Put, and K. De Bosschere. Understanding Obfuscated Code. In *14th IEEE International Conference on Program Comprehension (ICPC '06)*, pages 268–274. IEEE, 2006.

[139] A. Majumdar, A. Monsifrot, and C. Thomborson. On Evaluating Obfuscatory Strength of Alias-Based Transforms Using Static Analysis. In *International Conference on Advanced Computing and Communications (ADCOM 2006)*, pages 605–610. IEEE, 2006.

[140] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *European Symposium on Programming (ESOP)*, pages 5–20, 2005.

[141] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. A Taxonomy of Self-Modifying Code for Obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[142] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 1976. ISSN 0098-5589.

[143] J.M McCune, B.J. Parno, A. Perrig, M.K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[144] J.M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[145] W. Michiels and P. Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 82–89. ACM, 2007.

[146] W. Michiels, P. Gorissen, and H.DL. Hollmann. Cryptanalysis of a Generic Class of White-Box Implementations. In *Selected Areas in Cryptography*, volume 5381, pages 414–428. Springer, 2009.

[147] W.PR. Mitchell. Protecting secret keys in a compromised computational system. In *Information Hiding*, pages 448–462. Springer, 2000.

[148] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *28th IEEE Symposium on Security and Privacy*, pages 231–245. IEEE, 2007.

[149] A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In *23rd Annual Computer Security Applications Conference (ACSAC '07)*, pages 421–430. IEEE, 2007.

[150] T. Müller, F.C. Freiling, and A. Dewald. TRESOR runs encryption securely outside RAM. In *Proceedings of the 20th USENIX conference on Security*, pages 17–17. USENIX Association, 2011.

[151] M. Munson Taghi and C. John. Measurement of data structure complexity. *Journal of Systems and Software*, 20(3):217–225, 1993. ISSN 0164-1212.

[152] M. Myska. The True Story of DRM. *Masaryk Ujl & Tech.*, 3:267, 2009.

[153] C. Nachenberg. Computer Virus-Coevolution. *Communications of the ACM*, 50(1):46–51, 1997.

[154] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *26th IEEE Symposium on Security and Privacy*, pages 226–241. IEEE, 2005.

[155] J.Q. Ning, A. Engberts, and W. Kozaczynski. Recovering Reusable Components from Legacy Systems by Program Segmentation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 64–72. IEEE, 1993.

[156] P. O'Kane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *Security & Privacy, IEEE*, 9(5):41–47, 2011.

[157] E.I. Oviedo. *Control flow, data flow and program complexity*. McGraw-Hill, Inc., 1993. ISBN 0077074106.

[158] U. Piazzalunga, P. Salvaneschi, F. Balducci, P. Jacomuzzi, and C. Moroncelli. Security Strength Measurement for Dongle-Protected Software. *Security & Privacy, IEEE*, 5(6): 32–40, 2007.

[159] I.V. Popov, S.K. Debray, and G.R. Andrews. Binary Obfuscation Using Signals. In *Usenix Security Symposium*, pages 275–290, 2007.

[160] D.A. Quist and L.M. Liebrock. Visualizing Compiled Executables for Malware Analysis. In *6th International Workshop on Visualization for Cyber Security, 2009 (VizSec '09)*, pages 27–32. IEEE, 2009.

[161] J. Raber and E. Laspe. Deobfuscator: An Automated Approach to the Identification and Removal of Code Obfuscation. In *14th Working Conference on Reverse Engineering (WCRE '07)*, pages 275–276. IEEE, 2007.

[162] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[163] J. Riordan and B. Schneier. Environmental Key Generation Towards Clueless Agents. *Mobile Agents and Security*, pages 15–24, 1998.

[164] P. Röder, F. Stumpf, R. Grewe, and C. Eckert. Hades-hardware assisted document security. In *Second Workshop on Advances in Trusted Computing (WATC 2006 Fall), Tokyo, Japan*, 2006.

[165] R. Rolles. Unpacking Virtualization Obfuscators. In *3rd Usenix Workshop on Offensive Technologies (Woot '09)*, 2009.

[166] J. Rott. Intel Advanced Encryption Standard Instructions (AES-NI). Technical report, Technical report, Intel, 2010.

[167] K.A. Roundy and B.P. Miller. Binary-Code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 46(1):4, 2013.

[168] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *22nd Annual Computer Security Applications Conference (ACSAC '06)*, pages 289–300. IEEE, 2006.

[169] S. Rugaber, K. Stirewalt, and L.M. Wills. The Interleaving Problem in Program Understanding. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 166–175. IEEE, 1995.

[170] A. Saxena, B. Wyseur, and B. Preneel. Towards Security Notions for White-Box Cryptography. In *Information Security*, pages 49–58. Springer, 2009.

[171] S. Schrittwieser and S. Katzenbeisser. Code Obfuscation against Static and Dynamic Reverse Engineering. In *Proceedings of the 13th International Conference on Information Hiding*, pages 270–284. Springer, 2011.

[172] S. Schrittwieser, S. Katzenbeisser, P. Kieseberg, M. Huber, M. Leithner, M. Mulazzani, and E. Weippl. Covert Computation: Hiding Code in Code for Obfuscation Purposes. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 529–534. ACM, 2013.

[173] S. Schrittwieser, S. Katzenbeisser, P. Kieseberg, M. Huber, M. Leithner, M. Mulazzani, and E. Weippl. Covert Computation – Hiding Code in Code through Compile-Time Obfuscation. *Computers & Security*, 42(0):13 – 26, 2014. ISSN 0167-4048.

[174] E.J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Usenix Security Symposium*, 2013.

[175] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *9th Working Conference on Reverse Engineering*, pages 45–54. IEEE, 2002.

[176] A. Shamir and N. Van Someren. Playing 'Hide and Seek' with Stored Keys. In *Financial Cryptography*, volume 1648, pages 118–124. Springer, 1999.

[177] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee. Eureka: A Framework for Enabling Static Malware Analysis. *Computer Security-Esorics 2008*, pages 481–500, 2008.

[178] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *30th IEEE Symposium on Security and Privacy*, pages 94–109. IEEE, 2009.

[179] M.I. Sharif, A. Lanzi, J.T. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of Network and Distributed System Security Symposium (NDSS '08)*, 2008.

[180] A. Slowinska, T. Stancescu, and H. Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of Network and Distributed System Security Symposium (NDSS '11)*, 2011.

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

[182] H.M. Sneed. Encapsulation of Legacy Software: A Technique for Reusing Legacy Software Components. *Annals of Software Engineering*, 9(1-2):293–313, 2000.

[183] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M.G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security Via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote Invited Paper.*, 2008.

[184] Y. Song, M.E. Locasto, A. Stavrou, A.D. Keromytis, and S.J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 541–551. ACM, 2007.

[185] J. Stewart. Ollybone: Semi-Automatic Unpacking on IA-32. In *Proceedings of the 14th Def Con Hacking Conference*, 2006.

[186] G.E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, pages 9–14. ACM, 2007.

[187] O. Sukwong, H. S Kim, and J.C. Hoe. Commercial antivirus software effectiveness: an empirical study. *Computer*, pages 63–70, 2011.

[188] Y. Tang and S. Chen. An Automated Signature-Based Approach against Polymorphic Internet Worms. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):879, 2007.

[189] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed Proof Generation for Machine Code. In *22th International Conference on Computer Aided Verification (CAV '10)*, pages 288–305. Springer, 2010.

[190] S.R. Tilley, S. Paul, and D.B. Smith. Towards a Framework for Program Understanding. In *4th Workshop on Program Comprehension*, pages 19–28. IEEE, 1996.

[191] S. Tillich and J. Großschädl. Accelerating AES using instruction set extensions for elliptic curve cryptography. *Computational Science and Its Applications–ICCSA 2005*, pages 43–59, 2005.

[192] S. Tillich and J. Großschädl. Instruction set extensions for efficient AES implementation on 32-bit processors. *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 270–284, 2006.

[193] S. Tillich, J. Großschädl, and A. Szekely. An instruction set extension for fast and memory-efficient AES implementation. In *Communications and Multimedia Security*, pages 11–21. Springer, 2005.

[194] S. Treadwell and M. Zhou. A Heuristic Approach for Detection of Obfuscated Malware. In *IEEE International Conference on Intelligence and Security Informatics (ISI '09)*, pages 291–299. IEEE, 2009.

[195] H.Y. Tsai, Y.L. Huang, and D. Wagner. A Graph Approach to Quantitative Analysis of Control-Flow Obfuscating Transformations. *IEEE Transactions on Information Forensics and Security*, 4(2):257–267, 2009.

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

[197] S.K. Udupa, S.K. Debray, and M. Madou. Deobfuscation: Reverse Engineering Obfuscated Code. In *12th Working Conference on Reverse Engineering*, pages 10–pp. IEEE, 2005.

[198] A. Vasudevan, J. McCune, J. Newsome, A. Perrig, and L. van Doorn. CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 48–49. ACM, 2012.

[199] Z. Vrba, P. Halvorsen, and C. Griwodz. Program Obfuscation by Strong Cryptography. In *International Conference on Availability, Reliability, and Security (ARES '10)*, pages 242–247. IEEE, 2010.

[200] A. Walenstein, R. Mathur, M.R. Chouchane, and A. Lakhotia. Normalizing Metamorphic Malware Using Term Rewriting. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '06)*, pages 75–84. IEEE, 2006.

[201] C. Wang, J. Hill, J. Knight, and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical report, CS-2000-12, University of Virginia, 2000.

[202] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of Software-Based Survivability Mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, pages 193–202. IEEE, 2001.

[203] R. Weaver. A probabilistic population study of the Conficker-C botnet. In *Passive and Active Measurement*, pages 181–190. Springer, 2010.

[204] M. Webster and G. Malcolm. Detection of Metamorphic and Virtualization-Based Malware Using Algebraic Specification. *Journal in Computer Virology*, 5(3):221–245, 2009.

[205] H. Wee. On Obfuscating Point Functions. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 523–532. ACM, 2005.

[206] N. Wilde and M.C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 2006.

[207] C. Willems and F.C. Freiling. Reverse Code Engineering-State of the Art and Countermeasures. *IT-Information Technology*, 54(2):53–63, 2012.

[208] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *Security & Privacy, IEEE*, 5(2):32–39, 2007.

[209] M.J. Wolfe, C. Shanklin, and L. Ortega. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[210] 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.

[211] B. Wyseur. *White-Box Cryptography*. PhD thesis, KU Leuven, 2009.

[212] B. Wyseur and B. Preneel. Condensed White-Box Implementations. In *Proceedings of the 26th Symposium on Information Theory in the Benelux*, pages 296–301, 2005.

[213] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In *Proceedings of the 14th International Conference on Selected Areas in Cryptography*, pages 264–277. Springer, 2007.

[214] H. Yin and D. Song. TEMU: Binary Code Analysis Via Whole-System Layered Annotative Execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, 2010.

[215] J. Zeng, Y. Fu, K.A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation Resilient Binary Code Reuse through Trace-Oriented Programming. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013.

[216] Z. Zhao, G.J. Ahn, and H. Hu. Automatic Extraction of Secrets from Malware. In *18th Working Conference on Reverse Engineering (WCRE '11)*, pages 159–168. IEEE, 2011.

[217] X. Zhuang, T. Zhang, H.H.S. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 292–302, 2004.

[218] C.C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 138–147. ACM, 2002.

List of Figures

List of Tables