KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Electrical Engineering (ESAT)

Code Obfuscation Techniques for Software Protection

Jan CAPPAERT

Dissertation presented in partial
fulfillment of the requirements for
the degree of Doctor
in Engineering

April 2012

Code Obfuscation Techniques for Software Protection

Jan CAPPAERT

Jury:
Prof. dr. Adhemar Bultheel, chair
Prof. dr. ir. Bart Preneel, supervisor
Prof. dr. ir. Frank Piessens
Prof. dr. ir. Patrick Wambacq
Prof. dr. ir. Vincent Rijmen
Prof. dr. ir. Koen De Bosschere
(Ghent University)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

April 2012

D/2012/7515/34
ISBN 978-94-6018-498-7

Acknowledgments

This dissertation is the result of a long process to which many people were directly or indirectly involved. Therefore, I would like to take this opportunity to thank everyone who has supported me during the past years.

First, I would like to express my gratitude to Prof. Bart Preneel for giving me the opportunity to do research and pursue a PhD at COSIC, for his advice during those years, and for convincing me to write this dissertation. I want to thank Prof. Koen De Bosschere, Prof. Frank Piessens, Prof. Patrick Wambacq, and Prof. Vincent Rijmen for kindly accepting to be part of my jury, and Prof. Adhemar Bultheel for chairing it.

When I started working at COSIC, a new world opened for me. Although the group was a lot smaller then, I very much appreciated the warm welcome and the help from my colleagues to get me started. Therefore, I would like to thank all past and current COSIC members for contributing to this unique experience. In particular, I would like to thank the people that shared offices with me for creating a nice atmosphere, (not) watering my office plants, and organizing office BBQs.

This thesis would not have been possible without the collaboration with UGent/ELIS/CSL (former PARIS). Therefore, I would like to thank Prof. Koen De Bosschere for the invitations to his research group. I also would like to thank Matias Madou and Bertrand Anckaert for the fruitful discussions that led to several joint publications.

I would like to thank all persons that helped me during this final phase of my PhD. Andreas, thanks for proofreading my text, even after you admitted it was not your main area of expertise.

Péla Noë deserves a big thank you for many reasons. Not only for answering my numerous questions concerning paperwork, but also the informal chats have definitely contributed to my COSIC experience. Péla, bedankt!

Ik wil ook mijn ouders bedanken voor de kansen die ze me geboden hebben. Mijn ouders hebben altijd achter mijn studiekeuzes gestaan en me geholpen waar ze konden. Ik wil vooral ook mijn vriendin Sofie bedanken voor het geduld en voor de extra zorg die ze onze kindjes gaf tijdens mijn afwezigheid. Tot slot, Kobe en Korneel, wil ik jullie bedanken voor alle mooie en minder mooie momenten die steeds voor de nodige afwisseling zorgden.

Jan Cappaert
April 2012

Abstract

This thesis examines code obfuscation techniques to protect software against analysis and unwanted modifications. Program obfuscation makes code harder to analyze. Indirectly, this also contributes to protecting against malicious modifications to a program. This stems from the fact that an attacker first must understand the software before he can make specified modifications. In addition to techniques that improve a program's analysis resistance, one can add techniques that make tampering hard.

First, we present a comprehensive overview of software protection techniques. These are compared in terms of protection against analysis and protection against modifications. In both cases a distinction is made between protection against static attacks and protect against dynamic attacks.

The main part of this thesis describes two research contributions. A first contribution describes a technique which makes it difficult to statically derive the control flow of a program (this is the construction of the control flow graph). This technique is based on one-way functions to make backward analysis hard, and bijective functions that do not leak any control flow information that can assist an attacker. We continue by presenting three application models and several attacks to evaluate the strength of our technique.

The second contribution presents a technique to thwart dynamic attacks. These attacks include: (1) dynamic analysis of the code, for example by dumping memory, and (2) dynamic tampering. Against the first threat, we protect by introducing an on-demand encryption scheme which decrypts code just before its execution and re-encrypts it afterwards. Accordingly, exposure in memory is limited. Dynamic modifications are precluded by implicitly verifying code fragments and making other code fragments dependent on the outcome of this verification. We use cryptographic hash functions to hash code fragments; the resulting hash values are used as the decryption/encryption keys for other code fragments. This technique has been implemented using Diablo, a link-time

binary rewriter designed by Ghent University. To illustrate the cost of our technique, we protect several programs from the SPEC CPU2006 benchmark and measure the runtime overhead. Finally, we define a heuristic which identifies frequently executed code allowing us to trade off between our more secure on-demand encryption scheme and the more efficient bulk encryption.

There is an urgent need for flexible and cheap software protection techniques; moreover, deploying these techniques in untrusted environments is extremely challenging. We present some proposals for further research that help to address these challenges.

Samenvatting

Deze thesis bestudeert code obfuscatie technieken die software beschermen tegen analyse en ongewenste wijzigingen. In eerste instantie maakt code obfuscatie een programma moeilijker om te analyseren. Onrechtstreeks draagt dit ook bij tot de bescherming tegen het kwaadwillig wijziging van een programma. Deze knoeibestendigheid vloeit voor uit het feit dat een aanvaller eerst de software moet begrijpen alvorens hij specifieke wijzigingen kan doorvoeren. Naast de technieken die analyse bemoeilijken, kan men ook een aantal technieken toevoegen die specifiek gericht zijn op de knoeibestendigheid van software.

Allereerst geven we een uitgebreid overzicht van bestaande softwarebeveiligingstechnieken. Deze worden onderling vergeleken in functie van de bescherming tegen analyse en de bescherming tegen wijzigingen. In beide gevallen wordt er onderscheid gemaakt tussen bescherming tegen statische aanvallen en bescherming tegen dynamische aanvallen.

Het belangrijkste deel van deze thesis beschrijft twee bijdragen aan het onderzoeksdomein van softwarebeveiliging. Een eerste bijdrage bestaat uit een beschrijving van een techniek die het moeilijk maakt om statisch het controleverloop van een programma te achterhalen (dit is de constructie van de controleverloopgraaf). In deze techniek maken we onder andere gebruik van 1-wegsfuncties om achterwaartse analyse te verhinderen en van bijectieve functies die geen controleverloopinformatie lekken die bruikbaar is voor de aanvaller. Verder presenteren we drie modellen om onze techniek toe te passen en bespreken we enkele aanvallen om de sterke van onze techniek te evalueren.

De tweede bijdrage beschrijft een techniek om dynamische aanvallen minder succesvol te maken. Deze aanvallen bestaan uit: (1) dynamische analyse van code, bijvoorbeeld door het geheugen te inspecteren, en (2) dynamische wijzigingen. Tegen de eerste dreiging beschermen we door code vlak voor de uitvoering te ontcijferen en nadien te hervercijferen. Bijgevolg, zal deze minder lang blootgesteld zijn in het geheugen. De dynamische wijzigingen worden

tegengegaan door code impliciet te verifiëren en andere code afhankelijk te maken van het resultaat van deze verificatie. Hiervoor gebruiken we cryptografische hashfuncties die de code hashen en waarvan de resulterende hash gebruikt wordt als decryptie/encryptiesleutel voor het vercijferen van andere code. Deze techniek is geïmplementeerd met behulp van Diablo, een linktime binaire herschrijver ontworpen aan de Universiteit Gent. Om de kost van onze techniek te meten beschermen we verscheidene programma's uit de SPEC CPU2006 benchmark en meten we de vertraging in uitvoeringstijd. Tot slot definiëren we een heuristiek die voor frequent uitgevoerde code toelaat een trade-off te maken tussen veiligheid en performantie.

Aangezien er enerzijds een dringende nood is aan flexibele en goedkope softwarebeveligingstechnieken en anderzijds het zeer uitdagend is om deze te realiseren in een onbetrouwbare omgeving, sluiten we af met enkele voorstellen voor toekomstig onderzoek.

Contents

List of Abbreviations

AES	Advanced Encryption Standard
ASLR	Address Space Layout Randomization
ATM	Automated Teller Machine
CFG	Control Flow Graph
CPU	Central Processing Unit
CTR	CounTeR
DES	Data Encryption Standard
DRM	Digital Rights Management
ECB	Electronic CodeBook
ELF	Executable and Linkable Format
ICE	Integrity Checking Expression
IMR	Individualized Modular Redundancy
IP	Intellectual Property
IVK	Integrity Verification Kernel
MIPS	Microprocessor without Interlocked Pipeline Stages
MMU	Memory Management Unit
OCB	Offset CodeBook
OFB	Output FeedBack
OH	Oblivious Hashing
PC	Personal Computer
PCBC	Propagating Cipher-Block Chaining

RAM	Random-Access Memory
TEA	Tiny Encryption Algorithm
TRS	Tamper Resistant Software
TTS	Tamper Tolerant Software
USB	Universal Serial Bus
VM	Virtual Machine
VoIP	Voice over Internet Protocol
WBC	White-Box Cryptography

Chapter 1

Introduction

Information exchange has become an essential component in modern society. Vendors provide content to consumers, while consumers exchange information using e-mail, peer-to-peer systems, social networks, or other network applications. We rely on embedded software in our cars, we trust the domotics built into our homes, and we use electronic devices on a daily basis. Hence, the usage of software applications has become one of the corner stones of our lives. Obviously, all these applications rely on the correct functioning of software and hardware.

In the 1980s, application security was achieved through secure hardware, such as ATM terminals or set-top boxes. Since the 1990s, however, software protection has gained much interest due to its low cost and flexibility. Nowadays, we are surrounded by software applications, e.g. for online payments, social networking, games, etc. As a result, threats such as piracy, reverse engineering, and tampering have emerged. These threats are exacerbated by poorly protected software. Therefore, it is important to have a thorough threat analysis (e.g. STRIDE [65]) as well as software protection schemes.

In nowadays software era, the revenues of software companies are huge. Not only operating systems, but also professional applications (e.g. graphics software) can be very expensive. As a consequence, illegal use of software emerged. With just a few mouse clicks, people can download software, apply a downloaded patch to it, and start using it without payment. Vendors realized that protecting software against malicious users is a hard problem. The user is in control of his machine: he has physical access to the hardware, he controls the network connectivity, etc.

Nevertheless, software owners also manage to arm themselves against these

threats. Examples include popular applications such as Apple's media player iTunes, the voice-over-IP application Skype, or online games such as World of Warcraft. These applications have been exposed to attacks since years. Nevertheless, they still withstand the major problems caused by software threats such as reverse engineering, tampering, or piracy.

1.1 Problems of Software Protection

The last decade, more and more software gets distributed over the Internet. Examples range from browsers and e-mail clients to games, online shopping, e-banking, tax-on-web, and even e-voting. Once distributed to a client machine, the software owner actually loses all control of the (client) application. Furthermore, more software and platforms become mobile. Many wireless devices, including laptops, tablets, and smart phones, are becoming part of our daily lives.

Naturally, adequate security is required in this complex heterogeneous environment. First of all, software contains secret, confidential or sensitive information, for example medical files or credit card numbers. To protect this data, there exist encryption and authentication algorithms [94], but these require that secret keys have to be protected somehow. Secondly, we have the code of applications. It might be necessary to shield a certain software implementation from reverse engineering. And third, there is the program's execution itself, where critical code is executed and some confidential data is accessed. During execution one needs to protect the code and data from malicious intents, such as dynamic analysis and tampering. All these elements need to be sufficiently strengthened to guarantee data confidentiality and secure program execution to the user.

1.1.1 Malicious Software versus Malicious Hosts

Protection against malicious software. More than two decades ago it became clear that software had to be protected against malicious code. In 1988, some innocent programming issues turned Morris's harmless worm into "the Internet worm" [103], performing a denial-of-service attack that affected thousands of Unix machines. Programmers suddenly realized the potential impact of self-spreading code, especially on critical systems such as network servers.

Since then, the computer industry is focusing on how to protect hosts against malicious programs by for example developing virus scanners and firewalls.

Especially the last decade, this theme got additional attention because of the global impact of certain viruses and worms [73, 97] and last year the effectiveness of Stuxnet [148] was world news.

To protect against malicious programs, one will typically restrict the number of actions a client program is allowed to execute and/or shield his machine from the network. A simple but fast technique was to scan incoming code for signatures identifying viruses. The effectiveness of this technique relied on the number of false positives (innocent code marked as malicious) and false negatives (undetected malicious code). As a reaction, virus writers started rewriting virus code. They used code transformations, to disrupt pattern detection, and they used polymorphic code to avoid detection by static inspection of binaries. Namely, the virus body remained encrypted during spreading, while the virus decrypts its payload at runtime once a system is infected. Nowadays, virus scanners are large software applications offering multiple scanning techniques that allow a user to protect his machine from malware. To address the shortcomings of static signature scanning engines, *sandboxing* [100, 101] was introduced. Here the object runs in a protected environment under observation. If it shows similarities to a known piece of malicious software, a flag is raised, and countermeasures can be taken. The interaction between the virus writers and the anti-virus programmers could best be described as a software arms race.

Protection against malicious hosts. With the distribution of software towards client machines, owners lose control of their software. In this case, client programs are susceptible to attacks by a malicious host. This can be by a user with malicious intent or by other malicious software on the host, such as a Trojan horse. An example of this kind of attacks could be a user who is trying to crack a downloaded application. In this context, 'crack' can mean extracting secret keys, stealing intellectual property (such as algorithms), or making malicious changes in the code of the application, e.g. to use it without paying. Several techniques such as fault analysis [16] or searching for stored or embedded keys [116, 63] are easily applicable in this environment and very effective when programs are not additionally protected.

While protecting against malicious code or a malicious host are different problems, parties often use similar techniques to achieve the same goals. Malware authors typically want to preclude analysis and identification of their code. They achieve this by using obfuscation techniques. The same techniques can be used by legitimate programmers to protect their software against analysis. By doing this, they slow down a *global* attack on the software, i.e. an attack exploiting all

instances of an entire 'family' of programs. Figure 1.1 illustrates the four phases of this attack. First, an attacker needs to perform analysis to gain knowledge of the program internals, prior to tampering with it. When the tampering succeeds on a local program instance, the attacker might automate his attack such that it works also on other instances. And finally, the attacking code or cracked application will be distributed via the Internet, to have a global impact.

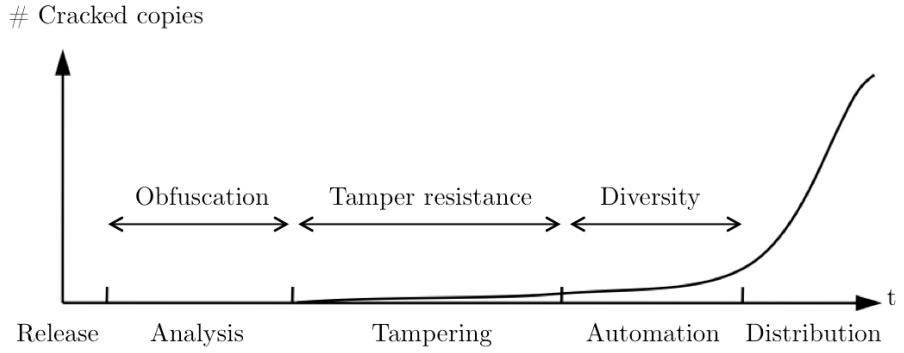


Figure 1.1: Cracking a software application typically implies in sequential order: analysis, tampering, automating the attack, and distribution of the 'crack' to launch a global attack.

1.1.2 Black-Box and White-Box Security Model

Historically, software ran on mainframes accessible by terminals. This was often denoted as *server-side execution* or *software as a service*. In this case the software itself is not distributed, but only the services. The user then communicates to a server which hosts the service and thus locally runs the software. In this case we speak about a *black-box model* because the user has no full privileged access to the server, but only access to the service.

As client machines became more powerful over the years, parts of a software application were delegated to the client. Hence, resulting in *partial client-server execution*. However, the main bottleneck was the network. The more interaction between client and server, the higher the network load. Nevertheless, software owners could still opt to run critical components at the server side, while non-critical parts were executed by the clients.

The major problem in the context of software security emerges when software is distributed to remote hosts. Once this is done, the owner loses all control over

the software. And from that moment on, malicious users or malicious software can abuse the local software, e.g. violating copyrights or license agreements. The model in which client software is attacked by local malicious software or by a malicious user is called a *white-box model*. In this model, the attacker has full access to the system on which the software is running. Chow *et al.* called these type of attacks *white-box attacks* [31, 30], because in this model the attacker has full privileged access to the system. This means that the malicious user (or program) can execute the program at will, he can observe the memory, processor, and registers, he change bytes during execution, etc.

1.1.3 Hardware-Based versus Software-Based Security

The main advantages of software-based protection techniques are the low cost and flexibility, e.g. its compatibility with existing systems. The main disadvantage is their limited strength. Due to evolving methods that circumvent application security, stronger techniques have to be invented [61]. Furthermore, techniques to either secure or attack software are sped up by increasing computing power of processors and growing capacity of storage media.

Local software and sensitive data can be secured by encryption and authentication. Whenever the software executes, it will decrypt its internals on the fly. Unfortunately, this method only guarantees full security if encryption and decryption is done in trusted hardware, for example on a cryptographic co-processor [131, 119]. But when this component is not available, as is the case on most existing systems, the problem becomes harder to tackle. Essentially, such a co-processor can be assumed to be a black-box system, where the attacker is only able to monitor input-output behavior. If encryption and decryption is not performed by trusted hardware, the program can be intercepted once it is sent in the clear to memory or processor. Additionally, *trusted computing* [4] offers a way to authenticate software and platforms. By relying on a small secure hardware component (e.g. a trusted platform module), this technique aims to verify the compliance of a user's system, i.e. enforcing both hardware and software to behave in expected ways.

A second hardware technique to protect software is tamper-resistant packaging. In this case the software and data are physically shielded from attacks. State of the art in these hardware techniques are cryptographic processors and smart cards. Nevertheless, even in this case some attacks can extract some information from these systems, namely by analysis of side channels such as time and power analysis [77, 76].

And a third example are hardware dongles. Hardware dongles are small hardware tokens (e.g. USB sticks) that need to be plugged into the computer when running

a protected software application. Without the dongle, the software will not run. One of the disadvantages is that dongles get lost. In that scenario the manufacturer has to create or issue a new dongle, or an entire new package, including a new dongle, has to be installed.

The two major disadvantages of hardware-based security technology are the high cost and the incompatibility with the current open computer platforms. By 'cost' we mean the expenses for buying and installing on the one hand, and the cost for upgrading and maintenance on the other hand. Hence, hardware solutions offer less flexibility than software solutions. If a mechanism has been broken, hardware needs to be updated. These updates are expensive and very slow to deploy. Therefore, software security research should be as hardware and platform independent as possible, so that it offers maximal flexibility. Nevertheless, we want to deliver the same amount of security as hardware solutions to resist practical attacks.

1.1.4 Legal versus Technical Countermeasures

Legal protection aims to create fear for an attacker, such that he would renounce his attack. *Patents* are obtained to protect intellectual property (IP), e.g. a proprietary algorithm. When a competitor uses a patent protected algorithm in his own product chain, the patent owner can sue the firm that 'stole' its IP. Hence, patents protect in a non-technical way against IP theft. Unfortunately, prosecution on a case by case basis is impractical and costly. Moreover, patents itselves are expensive and not always enforceable since it might be difficult to prove that a certain patent has been violated. Another example of IP are trade secrets. Secret valuable information, such as a formula, a design, a process, etc. could be considered a trade secret. However, these are not protected by law. As a consequence, they require extra protection.

In addition to patents, *copyrights* aim to prevent illegal copies of software. If end users install illegal copies on their machines, they risk a substantial fine. Again, this does not prevent piracy in a technical way, but it aims at increasing the fear to get caught and suffer the consequences.

License agreements typically describe what a user can do with a product, protected by a patent or a copyright statement. Examples include: statements about leasing the software, professional versus private use, etc.

1.2 Code Obfuscation and Tamper Resistance

Code obfuscation. *Code obfuscation* [41] is a set of program transformations that make program code and/or program execution difficult to analyze. First of all, obfuscation hinders manual inspection of program internals. By renaming variables and functions, and breaking down structures, it protects against reverse-engineering. It protects both storage and usage of keys, and it can hide certain properties such as a software fingerprint or a watermark, or even the location of a flaw in case of an obfuscated patch. However, code obfuscation itself does not protect from code lifting or software piracy. It merely strengthens built-in protection mechanisms, e.g. against tampering or piracy.

In theory, obfuscation could be used to hide vulnerabilities as well. However, *security through obscurity* is often not considered a good practice. Kerckhoffs' principle [74] states that a system should be secure, even if everything is known about the system, except the key. In the case of obfuscation, the 'key' can specify which transformations were performed, in what order, and on which section of the code. This key allows the software owner to reconstruct in a deterministic way a user's obfuscated, or even personalized, version of the binary. These keys can be kept in a database until required for maintenance or analysis of bug reports.

Tamper resistance. Software *tamper resistance* covers all techniques that make tampering with software harder. Obfuscation itself does not prevent tampering, but hinders the preceding analysis phase. As indicated in Figure 1.1, obfuscation can be a first step to protect against a global attack. In a global attack or class attack, multiple instances of an application are tampered with by exploiting the same security hole. For this reason, additional techniques need to be used to achieve tamper resistant software. Several papers propose self-checking code. Chang and Atallah [26] present "software guards" to detect and repair program changes, while Horne *et al.* [64] present a similar solution based on checksum functions. Aucsmith's scheme [11] presents techniques to achieve "tamper resistance software". His solution is built on the use of encryption, signing, and mutual verification of small portions of code.

1.3 Related Techniques

Below we summarize some techniques related to software protection. As they usually rely on assumptions such as analysis or tamper resistance, they are relevant to the field of software protection.

Software aging. A technique that discourages the use of older — possibly compromised — versions is *software aging* [68]. In this context, the value of an older version decreases compared to the newer version. However, newer versions are backward compatible with older ones, i.e. newer versions can read input of older versions but not vice versa. While at first glance, this technique might not seem to relate to security, it does rely on specific assumptions. For example, it assumes that attacking the newer version differs substantially from attacking the older version. Otherwise, attackers could simply port their attacks to the new version.

Software watermarking and fingerprinting. Software watermarking and fingerprinting do not prevent analysis, tampering, or piracy technically. However, similar to legal countermeasures, they aim to increase the attacker's fear for being caught. *Software watermarking* is an overall term that covers all techniques that prove ownership of software [41]. Hence, it protects against piracy. If a software watermark is unique per user, then it binds a program instance to a specific user. The latter calls *software fingerprinting*. In this scenario, it is possible to trace back a pirated copy to the 'traitor', i.e. the user who violated the license agreement by illegally distributing his copy.

Software diversity. Malware often attempts to exploit one or more flaws in commonly used operating systems or widely used applications. Certain software packages are vulnerable for these attacks and others are not, but due to the current "mono culture" in operating systems and Internet browsers, a single security flaw can cause major problems and have a worldwide impact [73, 97].

Similar to genetic diversity in nature, *software diversity* can protect a software community against self-spreading viruses and malware exploiting a common security vulnerability. Software diversity limits the impact of such a global attack, i.e. only a subset of the community will be affected by the attack.

1.4 Main Contributions

The main contributions of this thesis are as follows.

- A comparison of state-of-the-art software protection techniques.
- A model to protect against static control flow analysis.
- An implementation of a new type of software guard which offers both confidentiality and integrity of code.

First, while we are using cryptographic building blocks, our contributions do not offer perfect security. They rather shift the problem to the security or secrecy of a smaller component (similar to a key in cryptography).

Secondly, we make software parts depending on other parts. This strategy raises the bar for the attacker as he is forced to investigate a larger part of the application in order to attack a specific code fragment.

Finally, in this thesis, we focus on software-only solutions because of their low cost and flexibility. However, our practical solutions are structured in such a way that they can be easily mapped onto hardware-assisted protection schemes.

1.5 Outline of this Dissertation

This dissertation is structured in six chapters. After this introduction, Chapter 2 elaborates on software protection, threat models, and applications. Chapter 3 presents an overview and comparison of state-of-the-art protection techniques. Subsequentially, two chapters present contributions to the software protection field. Chapter 4 presents a model to strengthen control flow graph flattening such that it is resistant to local static analysis, while Chapter 5 examines a new type of software guards that protect software against both dynamic analysis and tampering. Both contributions rely partially on cryptographic primitives, such as hash functions and symmetric ciphers. Finally, we draw conclusions in Chapter 6 and suggest several directions for future research.

Chapter 2

Software Protection

2.1 Introduction

The need for software protection originates in multiple fields. *Data security* covers the confidentiality and integrity of data, typically during transmission and storage. However, it relies on a common assumption, namely that end points are trusted. Similarly, in the field of *network security* network components such as firewalls, access control systems, and intrusion detection systems assume (at least a number of) trusted hosts. However, fully trusted hosts are not straightforward, i.e. hosts or users might attempt to attack software applications.

Software security is the study of protecting software against analysis, tampering, and other means of exploitation. It aims to address the lack of trustworthy software in a untrusted host environment. Many variants however exist: stand-alone solutions versus network-assisted solutions (e.g. remote attestation), hardware-assisted solutions (e.g. using trusted platform modules, smart cards, or dongles) versus software-only solutions, etc.

We follow the view where applications consist of functional code (performing the actual program task), extended with other code (e.g. security code, error-handling code, etc.). Often one or more components are critical to the 'normal' execution of the program, i.e. correct and legitimate execution of the code. *Software protection techniques* serve as a binding to glue these critical components with the rest of the code into one monolithic application, whereas without software protection these types of code would have been easy to identify, isolate, and attack.

This chapter is structured as follows. Section 2.2 presents the threats to software protection, while Section 2.3 describes the three threat models in the world of software security. Section 2.4 presents some parameters to estimate the need for software protection, followed by a discussion on implementation issues. We conclude this chapter by discussing several application domains.

2.2 Threats to Software Applications

There are three major threats to software: reverse engineering, tampering, and piracy. Software reverse engineering represents techniques to inspect the inner workings of software applications, while tampering covers ways to tamper with software. Piracy concerns unauthorized use of software. On the one hand, legal countermeasures such as copyrights try to discourage the latter. On the other hand, technical protection mechanisms such as serial numbers and online registration effectively make it harder to duplicate protected software without detection, unless these protection mechanisms are tampered with.

In this thesis we do not focus on piracy prevention. We consider tampering and reverse-engineering the most basic attacks, the more so because piracy protection relies often on the same techniques that protect against analysis and tampering. Tampering attacks aim to change the functionality of the software while reverse-engineering techniques try to analyze the software. The following sections elaborate on software analysis, tampering, and piracy.

2.2.1 Analysis

Software reverse-engineering, or simply *software analysis*, mainly consists of investigating software. Depending on the intent of the attacker, one can extract hidden algorithms, secret keys, and other information embedded in the software. While analysis can be the actual goal of the attacker, it often is a precursor to tampering. Hence, hindering analysis also hinders tampering. In our attack model an attacker has full control over the host system and can therefore perform static and/or dynamic analysis techniques at will.

Static analysis. These techniques are applied on non-executing code and comprises static disassembling and subsequent static examination steps.

Disassembling code is usually done using either linear sweep or recursive traversal. *Linear sweep* simply scans over the code, disassembling instructions, assuming that every instruction is followed by another instruction. GNU's gdb [123] uses

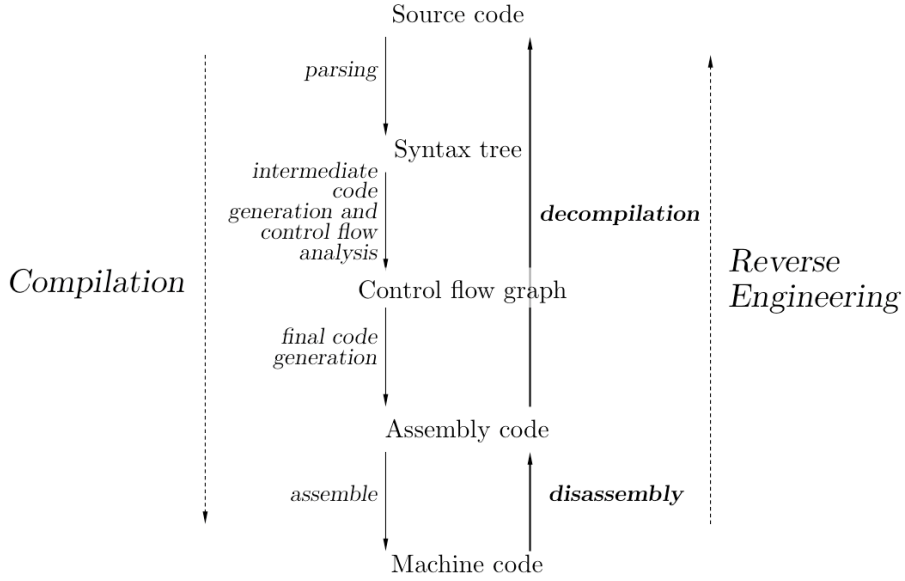


Figure 2.1: Different reverse engineering stages, represented as the inverse of a compilation process.

this technique. *Recursive traversal* takes control flow into account. However, as some branches are input-dependent, usually not all target addresses can be statically derived and disassembled. Linear sweep is easily fooled by inserted data bytes [84], while recursive traversal often gives incomplete results. As a reaction, Schwarz *et al.* [109] propose to use a combined approach. Additionally, Krügel *et al.* [78] treat every memory address as a potential start of an instruction. Consequently, they filter out overlapping assembly results based on a control flow graph based approach.

Additionally to disassembly, a decompilation step could map low-level code to more high-level constructions [33]. These high-level structures might facilitate human inspection more than abstract assembly listings. For certain languages such as Java or .NET, it is easy to decompile bytecode into source code [105].

Even if static analysis cannot derive all control flow, it is considered more complete than dynamic analysis as it examines all possible paths while dynamic analysis only considers the executed path. Figure 2.1 gives a schematic overview of the disassembly and decompilation phase for reverse engineering code. Recovered high-level code aims to approximate the original source code.

Dynamic analysis. Dynamic techniques are performed on executing code. This involves tracing of executed instructions, register contents, data values, etc. While this type of attack is more powerful than a static attack, it may be more time consuming or more complex. First, it requires a platform similar to that of the target code. Secondly, a program might be equipped with anti-debugging techniques hindering dynamic analysis. To this group of analyses belong debugging, emulation, etc. Commercial debuggers include SoftICE [20] and IDA Pro [49].

Hybrid analysis techniques. While static analysis gives very complete results when applied successfully, it is easily defeated by using self-modifying code such as code encryption or virtualization. Dynamic analysis, on the other hand, might be incomplete, or more time and resource consuming. As a response, researchers started combining both analyses. In the case of malware, code could be run in a software sandbox (e.g. a guest operating system) to examine its dynamic behavior. Resulting trace information could then be fed into static analysis tools, etc. Madou *et al.* [87] try to combine the best of both worlds by complementing statically gathered information with dynamically obtained control flow information. Similarly, Udupa *et al.* [132] illustrate that static control flow protection techniques lose most of their strength in a dynamic analysis. Debray and Patel [52] use offline analysis of dynamic execution traces to identify virus packers. And finally, two independent research teams both built a system to analyze virtualization-obfuscated software [117, 43].

2.2.2 Tampering

Tampering attacks go one level further compared to analysis attacks. Typically, one first needs information on the program internals before one can successfully tamper with the software. Therefore, tampering attacks are most often preceded by several reverse-engineering techniques. And once sufficient analysis has been performed, the attacker will try to change the software to meet his own needs. A hacker can make software available for everyone, even without authentication or rights to use the software. Similar to analysis, tampering can also be done statically or dynamically, this means on static or on executing code.

Static tampering. Static tampering techniques modify a static binary image. An *automated attack* consisting of downloading a crack and applying it on a stored binary is usually a static tampering attack, assuming that code is not loaded in memory and modified there.

Dynamic tampering. Dynamic tampering techniques modify an application at run-time. First, debuggers load code in memory. When stepping through instructions, one can modify code, data, or the state (e.g. register values) of the loaded program. A dynamic tampering attack is often performed "by hand" and has similarities to software debugging.

2.2.3 Piracy

Software piracy involves unauthorized use of software instances. During the last years, research has developed protection techniques to battle software piracy. First, *software watermarking* [41] aims at protecting software reactively against piracy. It embeds a unique identifier into an application such that it can be proven that a specific copy belongs to a specific individual or company. In the case where each software instance contains a unique identifier, it is often denoted as *software fingerprinting*. As a result, one can trace copied software to the source (i.e owner/user) unless the fingerprint is destroyed. Robust software watermarking schemes are still a topic for ongoing research. Often attackers apply code transformations which 'break' the software watermark, e.g. make it unreadable to the verifier (distortion), remove it, or add another watermark such that software can no longer be traced back to a unique user identity. Dynamic watermarks appear at runtime as a unique trace or data structure. In the case of "easter eggs", a special input is required to make the watermark visible.

2.3 Basic Threat Models

Threat models identify the threats to a system. They model both the attacker and the system to specify all possible attacks to a system. In the software community, we encounter three models, described below. As a summary, Table 2.3.3 gives an overview of these three main threat models.

2.3.1 Network Threat Model

In this model, as in reality, applications are network-connected. It covers browsers, e-mail clients, and other applications accessing the Internet. Here applications are assumed to reside on a trusted host. However, they are vulnerable to remote attacks. Hence, the attacker typically relies on pure input-output behavior of an application. He has no physical access to the trusted host other than via the running network application. This model is often referred to as the *black-box model*. However, note that an attacker might make assumptions

and acquire additional information about the remotely running application through other channels. He might even run a local copy of the application.

2.3.2 Insider Threat Model

Sometimes, an 'insider' has limited access to the trusted host running an application. He might be able to use the machine, however with limited privileges. As a consequence, and thanks to several attack tools, he might acquire information about the application (e.g. code or data). For example, by performing memory dumps and scanning memory for high entropy, one is able to steal confidential key material [116, 63].

Furthermore, *side-channel attacks* such as power, timing, or electro-magnetic radiation analysis fit into this model as attackers typically do no need full privileges on the machine. Just getting near a machine might already suffice for some of these attacks.

According to Aucsmith [11], Trojan horses and viruses belong to this model as well. Trojan horses might open back doors in software to grant access to adversaries, resulting in an attacker with limited privileges.

The insider threat model is sometimes referred to as the *grey-box model*.

2.3.3 Untrusted Host Threat Model

In this last model, the actual host itself is not trusted, nor is the user. Especially in private systems, but also in corporate environments, end users and their computers cannot be trusted.

First, their systems might be outdated, instable, or even compromised. In the latter case, an unauthorized user might gain superuser privileges. Secondly, the user himself might have malicious intent, e.g. the intent to violate the license agreement that comes with a software application. Intellectual property (data and/or code) within applications should not be extracted and stolen. Critical systems enforcing certain policies (access control, digital rights management, etc.) should not be tampered with. This model is widely known as the *white-box model* where the attacker has full privileges on the system. Furthermore, Aucsmith [11] divides this model into three subcategories:

- no special analysis tool;
- specialized software-based tools, e.g. using software breakpoints;

Table 2.1: The three main threat models for software applications and their distinctive properties. This thesis focuses on software security within the *white-box model*.

Threat model also called:	Network *black-box*	Insider *grey-box*	Untrusted host *white-box*
attacker's privileges	none	some	full
attacker's location	remote	local network or host	host
trusted host	yes	yes	no

- specialized hardware-based tools, e.g. emulators and bus logic analyzers.

White-box attacks [31] are hard to prevent on open systems, such as the PC, but software protection can raise the bar for the attacker. The work in this thesis focuses on this threat model.

2.4 The Need for Software Protection and Implementation Models

2.4.1 Estimating the Need for Software Protection

The level of security in an application consists of the required resistance of the application against reverse engineering and/or tampering attacks. By means of some model parameters, we can specify this level in more detail:

- Vulnerability: open systems are more vulnerable to attacks than closed systems. Desktops, notebooks, or mobile devices are more susceptible to white-box attacks compared to physically shielded servers accessible via a network connection.
- Value of content: the type of attacks and the amount of resources invested by an attack depend on the value of the content (code or data) and on the nature of the application.
- Content lifetime: content or properties with a longer lifetime require a higher level of trust and security.
- Security life cycle: the security of an application can be designed to be periodically renewable. Systems without upgrade possibilities need a higher security level than systems with systematic upgrades.

- Sensitivity for global attacks: global attacks are attacks concerning a whole group of software instances. This is feasible when code contains a "global secret". A single key for all legitimate users or an unpatched security flaw allows an attacker to develop an automated attack and spread it through the Internet.

The actual security level is always a trade-off between the need for software protection and the means to implement these software protection techniques.

2.4.2 Implementation Models for Software Protection

Software protection research steadily increased the last decade. In this section we describe several models in which code transformations are applied during different stages of software development.

Compilation of code has become more than just translating a computer program to an executable. Programs are often written in a high-level language because of two main reasons. First, the design can be done in an object-oriented way which facilitates programmers to maintain, adapt, and extend their code. Second, a rich high-level language typically offers a large set of standard classes containing predefined structures and functions. By using these classes, a programmer can save time and make sure that the called functions are optimally or securely implemented. Although this abstraction allows one to write more compact code, resulting programs might become large and slow due to the repetitive patterns resulting from object-oriented programming. This is why "code compilation" implicitly includes numerous optimization techniques varying from removing dead code, optimal register allocation, and efficient mappings to the target architecture's instruction set.

Figure 2.2 depicts our three levels to apply code transformations. For simplicity, we denote $T(x)$ as a protected instance of program x. However, $T()$ can be a combination of one or more program transformations implementing different software protection techniques. We distinguish the following three code transformation models:

- Model 1: pre-compilation or source-to-source transformations
- Model 2: at-compilation or code transformations during compilation
- Model 3: post-compilation or low-level transformations

In our model we clearly describe three distinct levels to apply obfuscation transformations, but combinations are possible as transformations can be applied

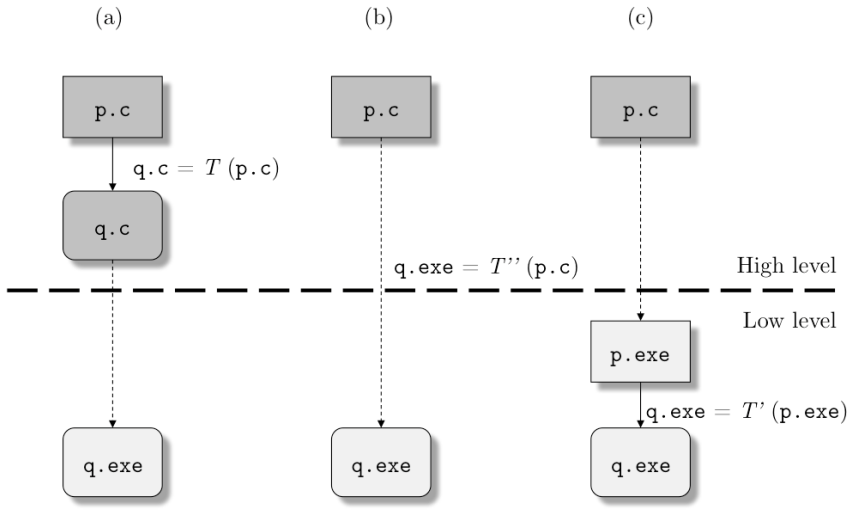


Figure 2.2: Where can code transformations can be applied? (a) At source code level, thus pre-compilation, (b) during compilation, and (c) low-level transformations, thus post-compilation. The dashed arrow represents the compilation process.

sequentially on a piece of code and at different levels during the code compilation process. The following sections elaborate on each of these levels where software could be modified and protected.

Source-to-Source Transformations

First, a programmer ships a self-made program, but he wants to add an extra layer of security to the most critical section in the program (e.g. authentication checking code). For C/C++ and other high-level languages, it is easy to write a tool that performs certain source-to-source transformations without changing the program's functionality. Typically, safe code transformations only affect code size and execution time.

Secondly, it is also possible to achieve binary obfuscation by first performing source-to-source transformations. Hence, it is not always necessary to apply additional binary transformation steps to applications designed in source code. It might be easier to apply certain transformation techniques on a high level, as source code is richer compared to binary code. In Madou *et al.* [86], we

perform high-level transformations, compile the target application, and observe the protection in the binary. Supported by empirical results, we conclude that several software protection techniques survive compiler transformations.

Code Transformations during Compilation

This model contains transformations that can be applied during the compilation process. To achieve this, one only needs to write a new compiler that is non-deterministic or that reads extra input determining where and how code should be transformed. If no extra input is given, the compiler itself should decide which transformations to use. In this way, we are able to generate diverse object code instances starting from one source code instance. Popular compiler frameworks include GNU's GCC, the open LLVM project, and Phoenix from Microsoft.

The general structure of a compiler is shown in Figure 2.3, although many compilers might skip or combine two or more phases into one module. We clearly distinguish several intermediate representations stating that a compiler has all knowledge of the internals of a program. While this knowledge is used for the translation process, it is as well processed to apply several code optimization techniques. A compiler performs some data and control flow analysis in order to build an interference graph for the register allocation. This flow analysis could also be useful for code obfuscation itself. For more details on compilers we refer to [5].

Observing the general scheme of a compiler, we examine where we could transform code resulting in more obscurity or diversity. This obscurity and diversity will obstruct code analysis and code decompilation. A possible phase to introduce diversity is instruction selection. Instruction selection translates expression trees into machine instructions. This process is called 'tiling' because multiple tile combinations can cover a single expression tree. Based on the flags set to the compiler, a specific mapping is chosen, e.g. the one optimal for speed. In this way a compiler generates programs, which look different at instruction level, but actually perform the same computations, as shown in Figure 2.4. Whether these resulting programs are harder to understand or more obscure than the original, will determine if this mapping is also useful for obfuscation purposes. Software metrics or heuristics could estimate how much both instances actually differ and how much extra effort an attacker has to do to reverse-engineer one of them. A compiler itself is actually useful as a tool for introducing code diversity and applying code transformations.

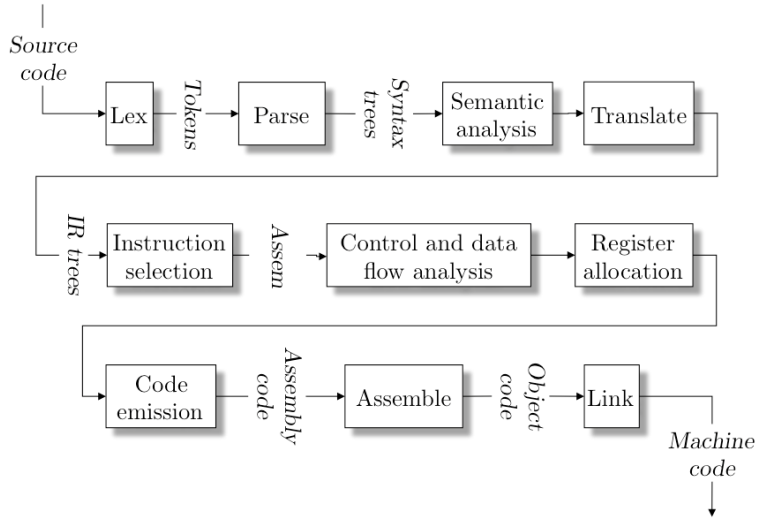


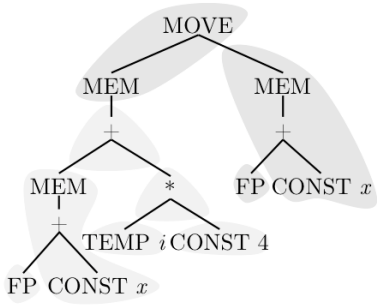
Figure 2.3: The general structure of a compiler. Current compiler designs might skip or combine several phases.

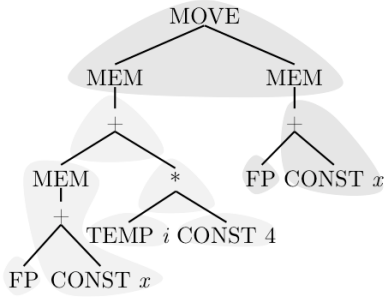
Low-Level Transformations

Similar to obfuscation at source level, we could perform transformations on a lower level. These transformations will thus take place after code compilation and we could therefore classify them as post-compilation transformations.

Compiling source code results in object code or some intermediate language code (e.g. Java bytecode). Although, the syntax of compiled code is simpler and less detailed than source code, it still leaks information, e.g. understandable instructions and procedures. Especially languages such as Java or .NET have a rich bytecode format. This often allows attackers to decompile the bytecode [105].

Tools such as ObjObf [110] and Burneye [128] protect object files. ObjObf allows to obfuscate ELF files, while Burneye also offers encryption to protect object files. ObjObf was based on extensive work from Wroblewski [141] who examined the potency of assembly code obfuscation.





LOAD	$r_1 \leftarrow \mathrm{M}[fp + a]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow r_i \times r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
LOAD	$r_2 \leftarrow \mathrm{M}[fp + x]$
STORE	$\mathrm{M}[r_1 + 0] \leftarrow r_2$

LOAD	$r_1 \leftarrow \mathrm{M}[fp + a]$
ADDI	$r_2 \leftarrow r_0 + 4$
MUL	$r_2 \leftarrow r_i \times r_2$
ADD	$r_1 \leftarrow r_1 + r_2$
ADDI	$r_2 \leftarrow fp + x$
MOVEM	$\mathrm{M}[r_1] \leftarrow \mathrm{M}[r_2]$

Figure 2.4: A tree tiled in two different ways. Both instructions sequences represent the same statement, but differ in the last two instructions due to the use of different 'tiles'.

2.5 Applications

Software obfuscation benefits many application domains. The bigger the need for security and diversity becomes, the more code transformations and other techniques will be utilized to protect critical data and code in applications. Software protection will be more and more integrated in the application and more often critical code has to be protected against white-box attacks. We sketch a few application domains below.

2.5.1 Digital Rights Management

A commercial application of software protection is Digital Rights Management (DRM), see also [31]. Namely, a DRM system has to enforce certain conditions to a user. Those conditions can be part of a license agreement which the DRM system has to enforce and thus, it requires adequate protection against tampering attacks.

Apple's all-in-one media player iTunes handles playing, downloading, and organizing music and video files. Originally, iTunes used the Fairplay as DRM technology. Downloaded songs could only be played on devices authorized by Apple. Furthermore, a song could only be played on a maximum number of devices linked to a single user. In 2005, Jon Lech Johansen released a program circumventing the Fairplay DRM, allowing to play downloaded music files on other devices [140]. Soon after this, a patch was released by Apple which got broken again by Jon. This typically illustrates the arms race between software protectors and attackers. In 2009, Apple changed its policy on DRM for music files. However, it still uses Fairplay for downloaded movies and Internet television.

Furthermore, because most software protection techniques allow to introduce 'randomness' in an implementation, instances can be individualized in such a way that they can be linked to the legitimate owner of that particular piece of software. This is the essence of watermarking and fingerprinting (see also Section 3.6.3).

2.5.2 Distributed Network Applications

Distributed network applications heavily rely on the correct functioning of client programs. If a user tampers with a program this might not only affect his own application, but in case of a distributed application it might also have impact on other connected clients.

In the context of gaming, users might tamper with applications in order to cheat. Each year, the gaming industry makes huge losses due to tampering or piracy. Both of these attacks can be hindered by anti-analysis techniques, such as code obfuscation.

Another example of a network application is Microsoft's Skype, a VoIP application that only allows Skype-authorized devices to connect to the network. In 2006, Skype was reverse-engineered [18]. It appeared that the application was heavily relying on multiple software protection techniques to avoid analysis and tampering. While not perfect, Skype's protection illustrates that software protection techniques raise the bar for an attacker aiming to reverse engineer or tamper with an application.

2.5.3 Mobile Software Agents

A mobile software agent is a program along with data, which is able to migrate from one computer to another autonomously and continue its execution on

the destination computer. In the case of mobile software agents, the user of the agent actually becomes the owner of the agent. Once the agent is given a task it can travel through a network, which can be an intranet, or even the Internet. The agent might carry valuable information from its owner that has to be protected during transmission, temporary storage, and execution of the agent.

Mobile software agents are used in multiple fields. Claessens *et al.* [34] investigate the feasibility of agents performing critical tasks on untrusted hosts. More specifically, the authors give an overview of mobile agent-based electronic transaction systems. In the field of trusted computing, Garay and Huelsbergen [57] propose to send a mobile agent to perform remote attestation (see also Section 3.6.1). In their work the authors propose an agent which is hard to analyze. Finally, D'Anna *et al.* [48] offer an extensive report describing challenges and solutions in state-of-the-art protection of mobile agents.

Chapter 3

Overview of Software Protection Techniques

3.1 Introduction

This chapter presents the state of the art for software protection techniques. We start by revisiting client server solutions and continue with software protection results from the 90s until now. These techniques are roughly ordered in categories. In the end we compare stand-alone software-based protection techniques with respect to the level of protection against static and dynamic analysis and tampering.

Our main contributions in this chapter include: (1) an overview of software protection techniques, with strong emphasis on software-only solutions, (2) a comparison of techniques judged on their potential to protect against analysis and tampering attacks, both static and dynamic.

3.2 Architectural Protection Techniques

We first look at software protection from an architectural point of view. These techniques usually do not fit in our *white-box model*, as described in Section 2.3.3. Nevertheless, we include them for completeness.

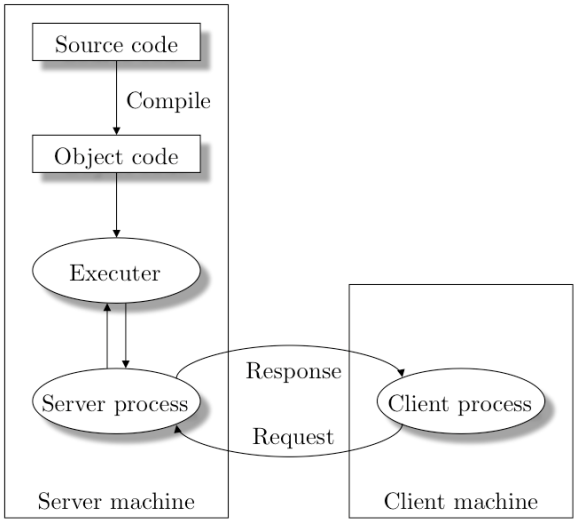


Figure 3.1: The client-server model only distributes access to services but not to the code, which is running at the server side.

3.2.1 Client-Server Solutions

One of the earliest techniques to protect critical software was to run it at the owner side instead of the user side, sometimes referred to as "software as a service". Critical software was not distributed to unreliable hosts, but maintained on a well protected server. The protection of the server depends on the combination of network, hardware, and software security (e.g. the operating system). The code itself is often not protected by any other techniques. By this setup, the services are distributed and not the software itself, as shown in Figure 3.1. Both the source code and the executable code will never leave the server side. From an attacker's point of view, the server will be seen as a *black box* that can be accessed by sending request and receiving responses.

The main disadvantage of client-server systems is that the server or the network bandwidth may become a bottleneck, causing services to be temporarily unaccessible. Although, this can be solved by mirroring and upgrading network infrastructure, a new model has been proposed, called *partial client-server* (see Figure 3.2). In this model, the sensitive code is split into a critical and a non-critical part. The critical part needs to be protected and is therefore run at the server side; the non-critical part is distributed and is run at the client side.

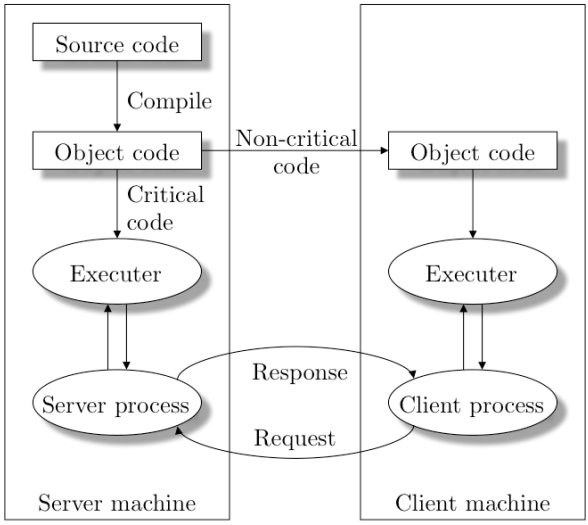


Figure 3.2: The partial client-server model splits code into a critical and non-critical part: the critical part is run at the server side; the non-critical part is run at the client side.

The advantage is that the load of the service is now distributed over the clients and the server. The code running at the server side can also be substantially smaller, although some extra overhead is needed to maintain communication between the client part and the server part. This directly indicates the main problem. At first glance, this model seems to unload the server, but in practice the client part and the server part require intensive communication so that once more the bandwidth becomes a bottleneck.

Although client-server was one of the first and still commonly used techniques to protect software from attacks, it avoids the problem by protecting the server and not the software running on it.

3.3 Techniques to Thwart Software Analysis

In this section, we present a number of state-of-the-art techniques that protect against analysis. Most techniques aim to obscure the inner workings of a program to protect against reverse engineering, statically or dynamically. Some techniques transform code offline, while others modify a program at runtime.

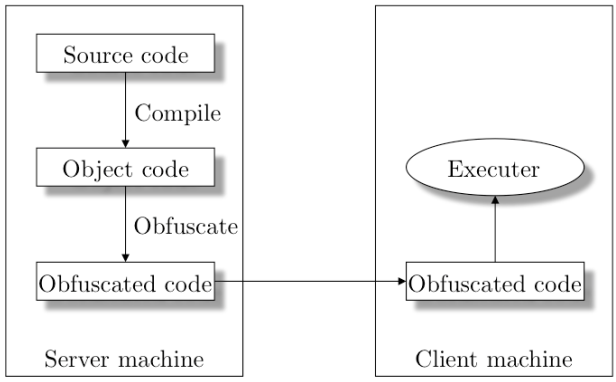


Figure 3.3: The code obfuscation model. Obfuscated code is fully run at the user side but has to be well protected against deobfuscation attacks.

Both classes of techniques raise the bar for an attacker who wants to perform adequate analysis, and consequently it will delay a tampering attack as well.

3.3.1 Collberg's Obfuscation Transformations

Object oriented programming is applied everywhere because it offers numerous advantages to read, adapt, or extend code. However, this way of programming in modules leaves many traces into an executable and reverse-engineers will exploit these traces to reconstruct the original source code [33]. When Java bytecode was shown to be susceptible to decompilation [105], yielding the original source code, researchers began investigating techniques to protect the code [37, 85]. Programmers have developed several techniques to maximally obscure the internals of a program so that analysis becomes very hard.

Collberg [41] defines an 'obfuscation' as a transformation that attempts to transform a program into an equivalent one that is harder to reverse engineer. In fact, *code obfuscation* applies one or more code transformations that make code more resistant to analysis and tampering, but preserve its functionality. Obfuscated code can then be distributed to untrusted hosts without risking to be reverse-engineered soon. This model is also drawn in Figure 3.3.

Code obfuscation was initially designed for languages such as Java because Java bytecode is very susceptible to code analysis. Many Java obfuscators [75, 120] (and deobfuscators) have been designed. Also .NET obfuscators [121] are

becoming common on the Internet. However, C/C++ obfuscators are harder to find, even if C and C++ are very common and widely used languages.

Wroblewski [141] and Mambo *et al.* [90] propose code obfuscation at the instruction level, e.g. assembly code. This has certain advantages. First, the code does not have to be compiled anymore, which facilitates the embedding of integrity checks and code hashing. This is one of the reasons why software guards [26, 64] are implemented at the assembly level. Second, transforming at the instruction level instead of at a high level is often preferred for watermarking [124].

Code Transformations

Several commercial code obfuscation programs only scramble identifier names and remove redundant information, such as debug information from the code. This is quite trivial, but obfuscation offers a lot more possibilities. A few large players in the software security world offer a full package of software protection techniques as a compiler infrastructure [81]. A good obfuscation is composed of one or more code transformations that transform a program's control and data flow such that it becomes harder to reverse engineer. The only restriction for these transformations is preserving the functionality of the original program. Thus, obfuscation is a collection of many techniques that are useful for program transformation, obfuscation or randomization.

Jakubowski *et al.* [70] call this "iterated transformations". For .NET, they present the Phoenix compiler and an analysis framework to modify intermediate code. Collberg's work mainly focuses on Java bytecode obfuscation, watermarking, and tamper proofing [41].

While code obfuscation techniques do not guarantee perfect security, a combination of several transformation techniques can lead to sufficient practical protection against reverse-engineering and tampering attacks. We consider practical protection as raising the bar for the attacker. If an attacker fails to analyze something due to a time limit, or a resource problem, the technique hindering that helps to practically protect the software. This is based on the need for security, (see also Section 2.4.1).

According to Collberg *et al.* [37], there are four main classes of code obfuscation transformations:

- lexical transformation;
- control flow transformations;

- data flow transformations;
- preventive transformations.

The following paragraphs elaborate on each of these transformations.

Lexical transformations. Swapping names of variables or replacing variable names by names without any semantic value are examples of lexical transformations. It is even possible to use heuristics to rename variables in order to mislead reverse-engineers. Many commercial tools focus on lexical transformations, while academic research focuses rather on the three other classes.

Control flow obfuscation. Control flow transformations alter a program's control flow (without affecting the overall functionality). A few important control flow transformations are:

- control flow graph flattening: Wang *et al.* [138, 137] describe a method for flattening control flow graphs (see also Section 4.4.1). This way control flow graphs adopt a uniform shape which is hard to analyze statically if equipped additionally with for example aliasing [79]. In Chapter 4, we present a technique that strengthens control flow flattening against static analysis.
- insertion of opaque predicates: Collberg *et al.* [42] present opaque predicates as the corner stone to add extra control transfers which will never be taken at runtime, but whose behavior is hard to deduce statically.
- Popov *et al.* [104] obscure control transfers by translating them into signals (traps) and inserting "junk" instructions after the signals.

Data obfuscation. Data obfuscation consists of several data transformations. Storage, encoding, aggregation, and ordering of data can be easily changed. Collberg *et al.* [38] describe data obfuscation as "breaking abstractions and unstructuring data structures". These data transformations range from obscuring inheritance relations, to table insertions (instead of a function), variable obfuscation, and constant obfuscation.

Most obfuscation transformations are not one-way. However, a few transformations are considered to be one-way. Sharif *et al.* [117] obfuscate `(x == A)` conditions by replacing them by `(hash(x) == B)` with `B` = $hash($`A`$)$. This

transformation is considered to be one-way because an attacker cannot statically deduce the original hard-coded `A`. Namely, if $hash()$ denotes a one-way function, then an attacker analyzing statically cannot deduce `A` from `B`. Note that this transformation can only be applied to data values and not to code.

Preventive transformations. All transformations that make specific deobfuscation techniques more difficult to succeed are called *preventive transformations*. Linn and Debray [84] fool linear sweep disassemblers by inserting "junk bytes" in between other instructions. The disassembly algorithm interprets these bytes as the beginning of an instruction, risking to incorrectly disassemble bytes of the original instruction succeeding the junk bytes. This is a clear example of an anti-disassembly trick. Similarly, anti-debugging techniques can be used to thwart debuggers and other dynamic analysis tools.

Obfuscation Metrics

Collberg *et al.* [37] and Low [85] present a method to evaluate obfuscation transformations. The authors define four quality metrics for an obfuscation transformation T:

- *potency* measures the extent to which T changes the complexity of a program P. This metric can for example be based on other software complexity metrics, e.g. McCabe's cyclomatic number [92];
- *resilience* measures how well a transformation T can resist an automated deobfuscation;
- *execution cost* is the time/space penalty a transformation T introduces to P;
- *stealth* measures how well the new code introduced by T blends into the application.

Subsequently, a metric *quality* of a transformation T is defined as a combination of the previous metrics to express how suitable a transformation T is:

$$T_{quality}(P) = \frac{(T_{potency}(P), T_{resilience}(P), T_{stealth}(P))}{T_{cost}(P)}$$

Anckaert *et al.* [10] propose to measure four program properties as metrics: instruction set, control flow, data flow, and data values. As an experiment, the authors apply obfuscation transformations such as flattening and opaque predicates to the SPEC CINT2000 test suite. The software metrics instruction count, cyclomatic number, and knot count were empirically evaluated and allowed to quantify the effect of the implemented obfuscation transformations.

Advantages and Disadvantages

The major advantage of software obfuscation is the low cost and the flexibility of this technique. Depending on the need for security (see also Section 2.4.1), an application can be obfuscated accordingly. In other words: the extra cost and computation time, introduced by the transformations, can be traded off with the performance.

The most important advantages of obfuscation are:

- Protection: obfuscation protects against static and dynamic analysis attacks as it raises the bar for the attacker, i.e. the attacker requires more time or resources to achieve his goal.
- Diversity: it is possibility to create different instances of the original program, e.g. to battle global attacks.
- Low cost: obfuscation involves a low maintenance cost due to automation of the transformation process and compatibility with existing systems.
- Platform independence: obfuscation transformations can be applied on high-level code so that platform independence is preserved.

The most important disadvantages are:

- Performance overhead: every transformation introduces an extra cost in terms of memory usage and execution time necessary to execute the obfuscated program.
- No long-term security: obfuscation by means of code transformations does not provide perfect security that makes analysis infeasible. Instead it makes program analysis harder, though not impossible.

The theoretical strength of obfuscation is indicated by the NP-hard results of Wang [137] and the PSPACE-hard results of Chow *et al.* [32]. Nevertheless, Barak *et al.* [13] point out that a "virtual *black-box* generator", able to hide code

of every program, cannot exist. However, this does not imply that protection by obfuscation for some programs is impossible.

Similar to encryption in practice never guarantees perfect protection, obfuscation cannot guarantee that an algorithm or the source code will never be extracted. Obfuscation rather aims to prevent that people or automated programs can decompile a program within a realistic, specified time.

State of the Art

Currently, numerous research results deal with code transformations, watermarks, and tamper resistant software [41, 11]. It is crucial that transformations neither change the program code, nor break the security code (e.g. a watermark).

There is a substantial expertise on Java bytecode obfuscation [37, 85] and on obfuscation of object code [141, 110], but there still is a lack of C/C++ obfuscation. Obfuscation of C/C++ code is limited to small 'obfuscation contests' [3], although C and C++ are very popular languages in the world of software development, e.g. in the open source community or for embedded systems. Furthermore, C/C++ has no additional built-in security mechanisms such as buffer overflow protection. Java is a type-safe language with built-in security mechanisms [1]. Hence, in the context of certain programming languages extra security measures are required during the development of critical software.

Obfuscation is possible at other levels as well:

- high level (source code), e.g. C/C++ of Java;
- intermediate level, e.g. Java bytecode;
- low level (object or machine code), e.g. assembly language.

In Madou *et al.* [86], we indicate that most obfuscation is applied to the target code itself: i.e. source-to-source transformations for source code protection, intermediate code transformation for intermediate levels, and binary obfuscation for binary protection. We empirically demonstrate that source code transformations and bytecode transformation can also lead to an obfuscated binary.

3.3.2 White-Box Cryptography

In the late 90s, attacks have been presented to extract key information out of RSA and even DES implementations. Boneh, Demillo, and Lipton [21] published a

method to extract keys from RSA. Biham and Shamir [16] introduced *differential fault analysis* to extract keys from secret key crypto systems, such as DES. As these particular attacks focus on the extraction of the secret key embedded in a cryptographic implementation, they became a new threat in security.

In August 2002, Chow *et al.* [31] defined the following new threat model:

- full privileged attack software shares a host with cryptographic software, having complete access to the implementation of algorithms;
- dynamic execution (with instantiated cryptographic keys) can be observed;
- internal algorithm details are completely visible and alterable at will.

The authors named this the *white-box attack context* or *malicious host attack context*. In this setup, the attacker's objective is to extract the cryptographic key. Obfuscation alone does not help against this threat, because obfuscated cryptographic algorithms still store parts of the secret key in the malicious host's memory which can be read by the attacker.

Chow *et al.* proposed a new technique to secure cryptographic algorithms against white-box attacks, called *white-box cryptography*. This technique is founded on the following ideas:

- transform a cipher (with a fixed key) into a series of lookup tables; and
- protect the key from extraction.

The latter is achieved by: (1) spreading out the key material over a network of key-dependent lookup tables instead of accessing it in a specific step, and (2) protecting each lookup table L_i by encoding its input and output. Hence, a lookup table L_i is replaced by a new lookup table $L_i' = G \circ L_i \circ F^{-1}$ in which F is an input encoding and G is an output encoding. The functions F and G allow to inject sufficient 'randomness' in the implementation so that locating and extracting the key material in a lookup table is hard. Still, the output encoding has to be canceled out by the input encoding of the next lookup table in order to preserve identical input/output behavior.

Given a cryptographic algorithm with embedded key, it can be transformed this algorithm into a series of lookup tables by using partial evaluation. This technique ensure that secret key information is contained in the resulting lookup tables. By insertion of mixing bijections, the secret key information is spread over all the lookup tables. Mixing bijections can be defined as a bijective transformations that maximize the dependence of the output regarding the

input. This means that a small change of input, even one bit, results in a maximum change in output.

Consider a series of lookup tables $L_1 \rightarrow L_2 \rightarrow L_3$, we can insert random bijections M_1, M_2 such that the new series of lookup tables represents $L_1 \rightarrow M_1 \rightarrow M_1^{-1} \rightarrow L_2 \rightarrow M_2 \rightarrow M_2^{-1} \rightarrow L_3$. We define the new lookup tables $L_2' = M_2 \circ L_2 \circ M_1^{-1}$ as the encoded version of L_2, respectively L_1' and L_3'. Applying this to the whole implementation results in a series of encoded lookup tables, that represent the same function, but that is locally secure. To achieve global security, an extra (global) input and output encoding is often proposed. This aims to prevent attackers from decoding the white-box implementation from the outside (the outer rounds) to the inside. Often, the first and last round of a cipher require less steps, making them more vulnerable to attacks.

Thus, the general idea is to spread the secret key information over the whole implementation, forcing an attacker to understand a greater part of the implementation. The current techniques are only applicable to cryptographic algorithms constructed with lookup tables, XOR functions, and permutations. This restriction is not a strong limitation as most symmetric cryptographic algorithms such as DES and AES [46] are constructed with these functions.

Advantages and Disadvantages

Currently, the advantages of white-box transformations are:

- Embedded key: secret key information is spread over the entire implementation of the cipher (i.e. the data representing the network of lookup tables), forcing an attacker to analyze the complete white-box implementation.
- Diversity: because of the randomness inserted into the implementation, we can create multiple diverse instances of one software application.
- Public-key application: if it is hard to extract the secret key, the white-box cipher can be used as a public-key tool. The cipher can be made public to encrypt, while its secret key remains secret. Attackers should either extract the secret key and feed it into a decryption routine, or invert table by table of the entire white-box implementation.

Unfortunately, most implementations still have serious disadvantages, mainly in performance:

- Execution: a significant increase in execution time, reducing the overall performance of the program.
- Code size: current lookup tables imply also a significant increase in code size.
- Security: the security of white-box techniques is still unknown and unproven.

State of the Art

White-box cryptography research started in 2002 with the publication of a white-box DES implementation by Chow *et al.* [31], soon followed by an AES implementation [30].

Shortly after the publication of the DES implementation, an attack was found by Jacob *et al.* [66]. Performance improvements have been proposed by Link and Neumann [83] and Wyseur and Preneel [146]. While the world gradually shifted from DES to AES, two groups of independent researchers cryptanalyzed the white-box DES implementation from Link *et al.* [145, 60].

Finally, the white-box AES implementation has been broken by Billet *et al.* [17] in 2004. Two years later, Bringer *et al.* [22] proposed a new white-box AES implementation where the S-boxes are part of the secret key. Last year, this version was cryptanalyzed by De Mulder, Wyseur, and Preneel [98].

For more details on white-box cryptography, we refer to Wyseur's PhD dissertation [144].

Applications and future research

Before the first white-box publications, it was believed protecting a cryptographic implementation against attacks in the white-box attack context was impossible because the executing malicious host has all the information concerning the algorithm and its execution. However, more and more publications shed light on the problems to solve.

Furthermore, researchers propose applications of white-box cryptography if proven secure and practical. Michiels *et al.* [95] use the tables of a white-box AES implementation as code to make it more tamper resistant. Namely, if code is tampered with, the lookup-tables will no longer be correct, and vice versa.

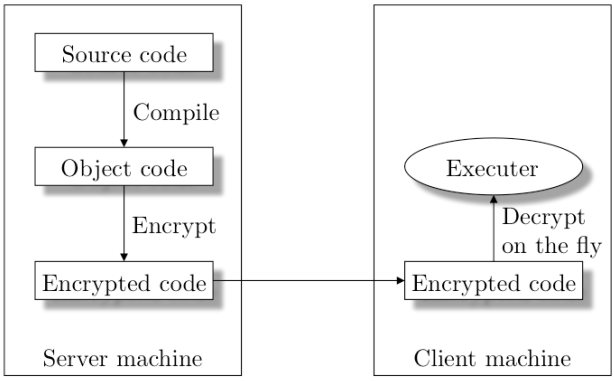


Figure 3.4: Encrypting code is done during transmission and storage. But once the code has to be executed it needs to be decrypted.

At this moment, it is still unclear if white-box cryptography can provide practical security in a white-box context. Researchers keep investigating the existence of a white-box resistant cipher and its applications.

3.3.3 Code Encryption

Tools such as *cryptographic wrappers* encrypt the code of a software application in order to avoid attackers gaining access to the code. This technique protects software against static reverse-engineering and tampering attacks as an attacker cannot inspect the code and therefore not make any structural changes when the code is stored on a disk or transmitted over a network. Note that an attacker can always flip random bits. However, when using non-malleable encryption it is not straightforward to obtain specifically altered plaintext by changing its ciphertext. Figure 3.4 illustrates the concept of code encryption.

During program execution parts of code will be decrypted "on the fly" with a secret key. Unfortunately, at that moment the code appears in the clear, in memory, where it can be intercepted. The intercepted code can then be debugged, decompiled, etc. This is the main vulnerability of this technique.

Even if the code or the data remains encrypted (see also Sander and Tschudin [107]), an attacker can still observe what happens during runtime if bits in the encrypted code or data are flipped. In the cryptographic community this technique is also known as *fault analysis* [16].

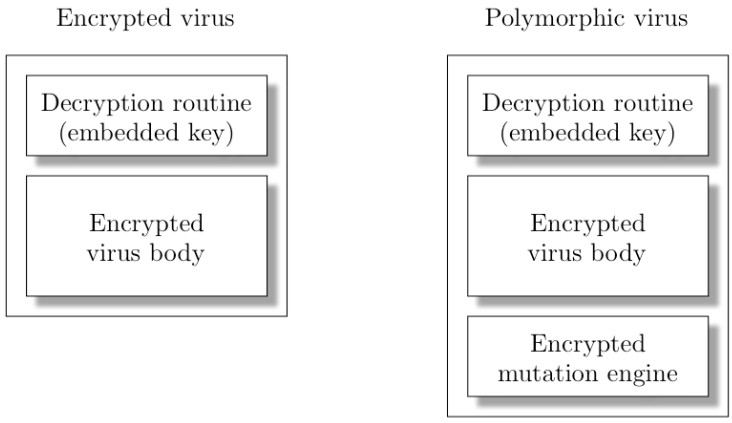


Figure 3.5: Encrypted viruses and polymorphic viruses encrypt their virus body to avoid signature detection. Polymorphic viruses even transform the decryption routine itself.

Encrypted and polymorphic viruses [125, 126, 147] perform code encryption techniques as well. An encrypted virus encrypts at each new generation the body with a unique key. This is mainly to avoid being detected by anti-virus engines scanning for virus signatures. A decryption routine is added to ensure that the virus body gets decrypted on the fly during execution. Nevertheless, if the encryption routine remains unchanged, scanning for signatures is still possible. For that, encrypted viruses evolved and added a mutation engine ensuring that for each new generation the decryption routine changes as well. The latter are therefore called polymorphic viruses. Note that the decryption routine can of course be protected with other techniques preventing analysis. Once a virus is decrypted and stored in memory, it will choose a new key, encrypt the new variant and add a modified decryption routine. Figure 3.5 shows the main structure of those viruses.

Shiva and Burneye are both ELF executable encryptors. Burneye [128] uses RC4 or a linear feedback shift register to encipher code, while Shiva opted for TEA, the Tiny Encryption Algorithm, optionally in combination with AES encryption. Cryptexec [135] uses CAST in ECB mode, while the extension CSPIM [136] adds a virtual machine to it. CSPIM first translates target code into MIPS instructions, which then are encrypted using RC5 in ECB mode.

In addition to self-encrypting code, UPX [102] uses compression to reduce the size of an executable, e.g. for systems with limited resources. While at first sight

this does not relate to security, UPX is used by viruses as well to 'obfuscate' their code. This technique achieves similar properties as encryption: rendering code unreadable. Another approach in reaction to the random properties of current packing techniques are mimic functions. Wu *et al.* [142] propose 'mimimorphism' and present a scheme that allows them to replace code by other code which conforms to a specific instruction distribution. Hence, the obfuscated code code mimics (statically) other code. At runtime, of course, the original code is reconstructed.

Code packing techniques encrypt the body of an executable, preventing static analysis of the binary. Consequently, this indirectly prevents static tampering as well. In Chapter 5 we propose the use of code encryption at a function level, in combination with a self-checking mechanism, built on top of the function call stack. These dependencies make on-the-fly tampering harder and avoid embedding of encryption keys.

3.3.4 Other Self-Modifying Techniques

Kanzaki *et al.* [72] describe how to overwrite program instructions with dummy ones. Hence, software should be crafted such that each 'hidden' instruction is restored prior to executing it. Similarly, Madou *et al.* [88] propose a "rewriting engine" that updates function bodies at runtime. Additionally, the authors propose to 'cluster' similar functions into a common template. Finally, Mavrogiannopoulos *et al.* [91] classify self-modification techniques based on the attacker's toolset capabilities:

- a disassembler;
- a debugger incapable of handling self-modifying code;
- a debugger that handles self-modifying code; or
- specialized tools.

Anckaert *et al.* [9] focus on hiding the actual data locations during execution. Their technique uses: (1) periodic reordering of the heap, (2) mitigating variables from stack to heap, and (3) pointer scrambling. Their system assumes a trusted software-based memory management unit which permutates memory pages at runtime.

3.4 Tamper Resistance Techniques

Tamper resistant software requires very skilled programmers working on a binary or source code level to embed "booby traps" for tamper detection in software. As modification of code is impossible to prevent on open platforms, we define tamper resistance as follows. *Tamper resistant software* aims to prevent tampering of a program by: (1) detecting undesired changes, and (2) reacting in case of tampering. Note that 'reaction' can also be more passive, e.g. a program that crashes after 'detecting' modification attempts.

3.4.1 Code Signing

Some languages (for example C) have no security mechanisms that check code before execution. Therefore, these languages are susceptible to tampering attacks changing the program in a way that its computations can no longer be trusted.

To avoid tampering with a program, its code needs to be protected during transmission and storage. Each time the program executes, it should check and verify its integrity to detect tampering. Signing techniques [94] are most suitable for this type of checking. The owner can sign the software and the user can verify the signature appended to the software. This model is shown in Figure 3.6. This is already the case with Windows drivers that are signed by Microsoft and verified by the operating system at installation time [96]. One could extend and automate this process so that the signature is verified at each execution of the program.

The main disadvantage is that verification of the signature relies on a public-key infrastructure in order to verify the authenticity of the public key. If not the case, an adversary could forge a signature.

3.4.2 Aucsmith's Tamper Resistant Software

In '96, Aucsmith [11] wrote one of the first papers defining "tamper resistant software" (TRS). He proposed a tamper resistant software architecture which bundles many techniques in order to realize a tamper resistant software implementation which can resist attacks of specialized software analysis tools. His technique combines four principles:

1. disperse secrets in both time and space;

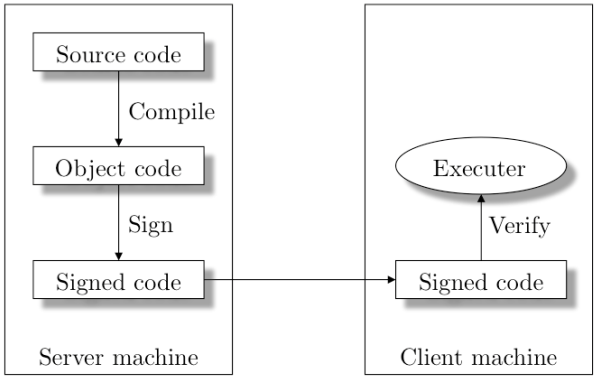


Figure 3.6: Code signing allows a user/program to verify if the code is coming from the signing identity and to decide whether it has been tampered with.

2. obfuscation of interleaved operations;
3. installation of unique code; and
4. interlocking trust.

One or more of these principles have later on been used as a basis for other protection techniques such as code diversity, software guards, code obfuscation, etc.

Aucsmith's architecture consists of two parts namely *integrity verification kernels* (IVKs) and an *interlocking trust mechanism*. An IVK is a small, armored segment of code to be embedded in a larger program. The IVK has two main functions:

1. stand alone and verify the integrity of code segments; and
2. communicate with other software in order to verify its execution.

The general structure of an IVK is shown in Figure 3.7. It is organized in cells, which are decrypted at runtime and thus determine the smallest amount of data/code which is ever exposed unencrypted. The encryption of cells is done in a pseudo-random order based on a generator function. Furthermore, each IVK contains one or more keys, including a secret key to sign and a public key to verify signatures made on other code segments.

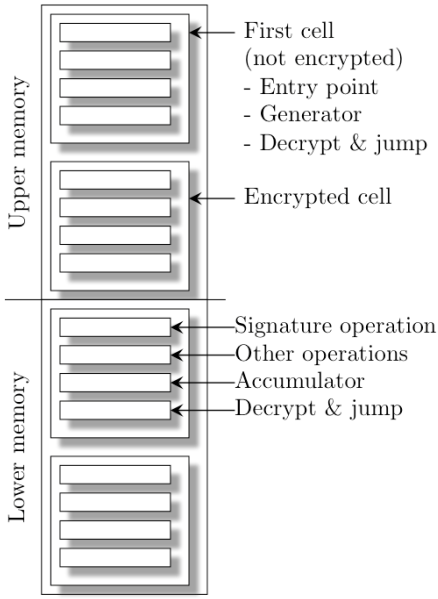


Figure 3.7: General structure of an integrity verification kernel (IVK), taken from [11].

The second part of the TRS architecture is the interlocking trust mechanism. It consists of IVKs, an integrity verification protocol and a system integrity program. These three components work together in an interlocking trust mechanism based on mutual integrity verification. An example is illustrated in Figure 3.8. For more details about the IVK and the interlocking trust mechanism we refer to [11].

3.4.3 Software Guards

Chang and Atallah [26] define small pieces of code that compute a checksum over code fragments. Calculating an integrity checksum can be done by for example CRC [139]. Using a complex, nested network, these *software guards* are able to verify each other's code plus the program code itself. In Chang and Atallah's scheme guards can repair 'damaged' (i.e. modified) program code. Hence, strictly speaking, a guard that repairs code at runtime is no longer a purely static technique as code gets modified at runtime. Aucsmith [11] calls mechanisms that "fight back" *active defense*. Anti-debugging techniques are

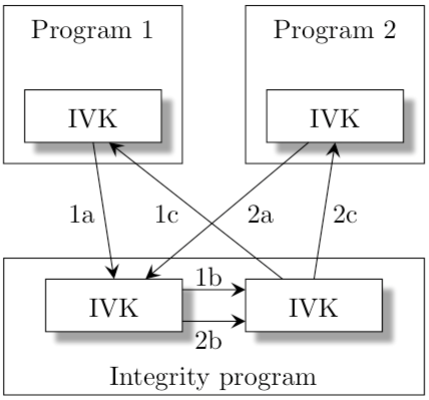


Figure 3.8: Verification of two programs through the interlocking trust mechanism.

another example of active defense. If a guard code can repair other guards as well, tampering with a program requires attacking the complete guard network. This implies identifying, localizing, and eliminating the whole network of guards and then tampering with the actual program code itself. A guards graph and its placement in a control flow graph (CFG) is shown in Figure 3.9.

The disadvantage of software guards is that it is hard to automate their construction and thus their strength depends mainly on one's programming skills. As a consequence, the maintenance cost will be very high.

New research from Horne *et al.* [64] tries to extend and automate this technique to improve tamper resistance of programs. Their technique is based on *testers* and *correctors*. The testers, in assembly language, are inserted at source code level, while the correctors are inserted in the object code. The values of the correctors and some watermark values are computed at installation time, resulting in a watermarked, self-checking, fully functional program.

Four years later, Wurster, van Oorschot, and Somayaji [143] exploited hardware to launch a successful attack on all self-checking software. Namely, it turns out that in many modern processors, *reads* to code and *fetches* of code can be distinguished. If an attacker duplicates memory pages and tampers with one version, he can redirect the reads to the untampered version, while the code fetched will be the tampered one. Despite this negative result, Giffin *et al.* [133] illustrated that self-modifying code is able to detect such a tampering attack.

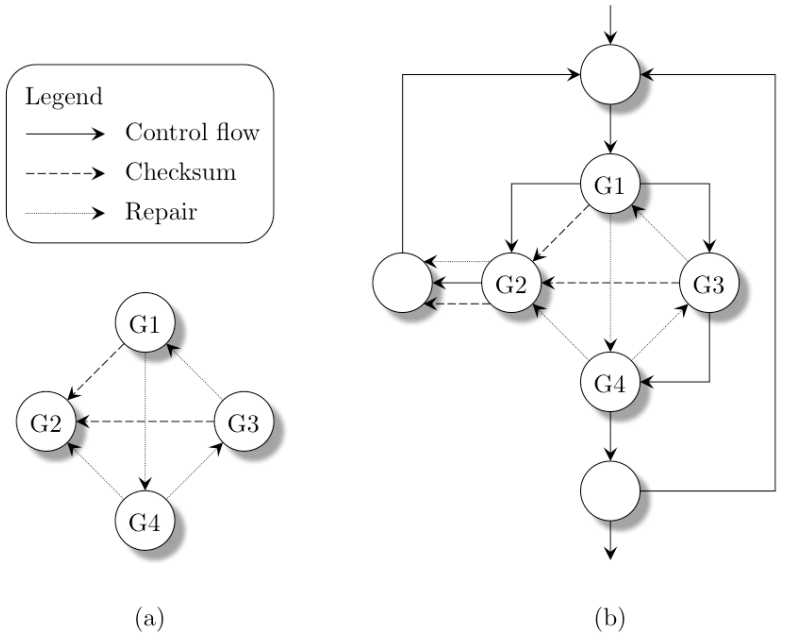


Figure 3.9: (a) A graph of guards and their actions. (b) A possible mapping of the guard network to a control flow graph.

Hence, it can trigger a reaction mechanism.

3.4.4 Oblivious Hashing

As a reaction on the concept of software guards checking only static code, Chen *et al.* [29] propose oblivious hashing (OH), a technique that allows implicit computation of a hash value of the actual execution. The idea is to hash the execution trace of a piece of code, allowing to verify the runtime behavior of the software. Hashing instructions are interweaved with the original code. They take results of previous instructions and apply them to hash values stored in memory (see Figure 3.10). As assignment results and control flow results capture most of the dynamic behavior of a program, it is sufficient to only hash assignments and control flows.

Oblivious hashing has two major application domains. First, it can be used to provide local software tamper resistance. Chen *et al.* [28] apply oblivious hashing

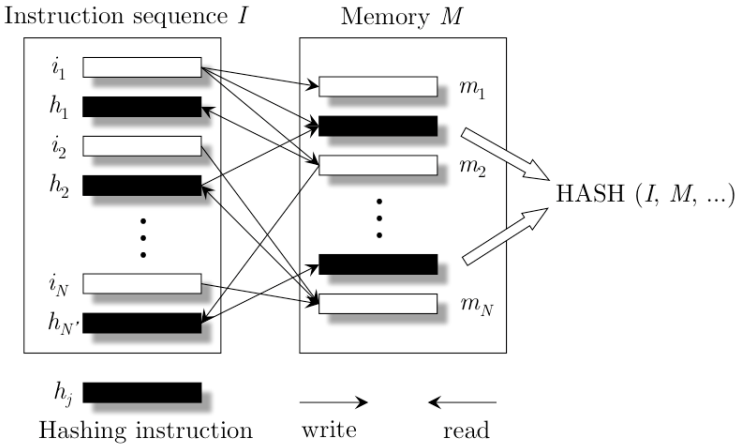


Figure 3.10: Oblivious hashing instructions are interweaved with original code and hash results of previous instructions in memory.

to Java bytecode to protect the call stack. Interweaved hashing instructions can inspect the top stack frame in order to verify whether the executing function is legitimate. Second, oblivious hashing can be used for remote code authentication. However in a white-box model, local software should provide its own security so that remote code authentication is usually not an option. In 2007, Jacob *et al.* [67] presented new work using oblivious hashing to make it more tamper resistant. In their paper, they complement OH with a combination of other techniques:

- overlapping of instructions;
- interleaving of instructions and code outlining;
- opaque predicates; and
- branch functions and junk instructions.

In addition, Jakubowski *et al.* [69] present special program predicates to check the integrity of a program, named integrity checking expressions (ICEs). These expressions dynamically check whether a program is in a valid state and act as a generalization of oblivious hashing techniques. Two candidates have been proposed:

- probabilistic verification conditions: these represent a reduced set of polynomials representing a program's intended behavior;
- Fourier learning approximations: this turns a program fragment F into a table of Learned Fourier coefficients such that these coefficients can be used to compute F.

3.4.5 Virtualization

Ghosh *et al.* [58] propose a secure and robust approach to tamper resistant software using process-level virtualization. The virtual machine (VM) acts as a just-in-time decryption routine. Furthermore, the VM periodically discards all previously decrypted code by flushing a cache containing decrypted code. Their technique also implies continuous shifting of application code to avoid dynamic attacks, and stealthy mechanisms that trigger flushing. The construction uses:

- virtualization: a VM decrypts code and periodically flushes its code cache;
- code encryption: code blocks are encrypted (see also Section 3.3.3);
- continuous shifting of code.

As virtualization typically shifts the problem, i.e. the VM becomes a possible target for attacks, additional self-checking mechanisms verify the integrity of the VM. Figure 3.11 illustrates that the software owner needs (knowledge of) the VM in order to create the intermediate code. The end user requires both the VM and the intermediate code to run the actual program.

3.4.6 Tamper Tolerant Software

Active mechanisms such as the repair functionality of Chang and Atallah's software guards (see also Section 3.4.3) imply redundancy in the code, i.e. repair mechanisms need to know what the original code was. Jakubowski *et al.* [71] propose Tamper Tolerant Software (TTS), using the notion of *individualized modular redundancy* (IMR). This technique implies redundancy of blocks on several granularity levels. Based on three building blocks (code redundancy, software individualization, and checkpointing – the ability to rollback at runtime), they implement the following properties in their protection scheme:

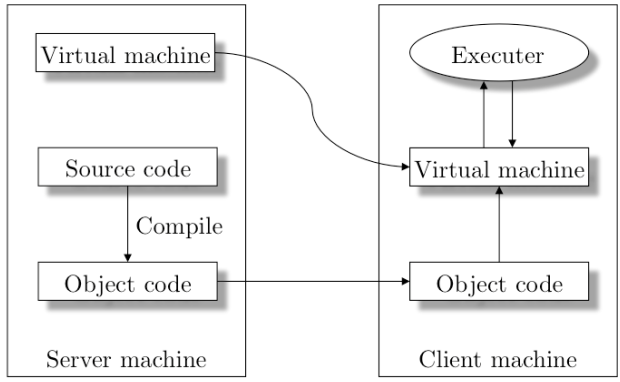


Figure 3.11: In case of virtualization, an application is split into a VM and object code. Both software parts have to be sent to the client machine in order to run the application.

- IMR with voting: similar to N-version programming [45] where N versions of a function vote for the result. The final result represents the winning vote, i.e. the majority vote.
- IMR with detection and correction: upon detection of tampering the block either runs an individualized copy of the block, or repairs the block prior to executing it.
- IMR with randomized execution: at runtime individualized blocks are chosen randomly for execution yielding that a tampered block has a probability $1/n$ to get executed in the case of n individualized copies.

To implement this protection technique, a program is chopped into blocks, which are duplicated, individualized, and rearranged. Appropriate logic to implement IMR schemes is added. Optionally, code for delayed tampering responses, checkpointing, or other techniques are added. To evaluate the effectiveness of TTS, the authors propose a graph-game model. For more details, we refer to [53].

Table 3.1: Approximation of technique against a specific type of attack. N = None, P = Partial, F = (almost) Full.

	Protection against			
	analysis		tampering	
Technique	static	dynamic	static	dynamic
Collberg's obfuscation transformations	P	P	P	P
White-box cryptography	F	P	F	F
Code encryption	F	P	F	P
Code signing	N	N	F	N
Aucsmith's tamper resistant software	F	P	F	F
Software guards	N	N	P	P
Oblivious hashing	N	N	P	P
Virtualization	P	P	P	P
Tamper tolerant software	N	N	P	P

3.5 Overview and Comparison

In this paragraph, we rate all the methods discussed so far according to the protection offered against analysis and tampering attacks, both static and dynamic. We did not include architectural protection techniques because it circumvents the problem of local software protection. Nor do we compare to related techniques (see Section 3.6) such as "code diversity" because this protection model is not focusing on a single software instance, but rather on a population of software instances. Table 3.1 gives an overview of the software protection techniques we described in the previous sections. All techniques are compared in terms of protection against analysis resistance and tamper resistance, both static an dynamic.

3.6 Related Techniques

In this section we elaborate on related techniques. Remote attestation relies on similar checking techniques as static/dynamic tamper resistance uses. The only difference is twofold: (1) techniques often assume presence of a trusted hardware component, and (2) actual verification is done at the server side.

Furthermore, this section elaborates on software diversity, watermarking, and buffer overflows. All previously discussed techniques allow one to individualize applications. This not only leads to software diversity, it can also serve as a

method to watermark software, and it also makes buffer overflow attacks harder, as these attacks rely on assumptions regarding the target code.

3.6.1 Remote Attestation

Vendors would like to verify the authenticity and correctness of client machines as for architectural solutions where a server is trusted (see Section 3.2.1). Remote attestation allows trusted third parties to remotely verify the state of a computer, i.e. the installed and running software on that machine. If unauthorized changes to the software have been made, remote attestation should detect these modifications and inform in some way the third party such that countermeasures can be taken. While this scenario is somehow different to the *white-box attack context*, the remote part is also vulnerable to white-box attacks. Hence, protection techniques are often related to those used in other solutions.

As it is hard to rely on the correctness of software on an untrusted host, remote attestation sometimes relies on a trusted platform module (i.e. a hardware module). However, Seshadri *et al.* [113] describe a remote attestation solution for embedded devices, without the need for a trusted platform module. Later, they proposed an adapted solution for legacy PC systems, called Pioneer [112, 111]. It consists of a two-stage challenge-response protocol. First, the verifier obtains an assurance that a verification agent is present on the untrusted host. Next, this verification agent reports the integrity of the executable the verifier is interested in.

When an adversary attempts to produce a correct checksum while running tampered code, this should be detectable due to an execution slowdown. When a memory copy attack [133] is deployed, an execution slowdown is caused by incorporating the program counter value and/or the data pointer value into the checksum computation. Because an adversary needs to forge these values, this will lead to an increase in execution time.

However, the design of the checksum function in Pioneer was subject to several constraints:

- The checksum function needs to be very efficient. If an adversary would be able to optimize the checksum function, he would gain time to perform malicious actions.
- To maximize the adversary's overhead, the checksum function will read the memory in a pseudo-random traversal. This prevents the adversary from predicting the memory reads beforehand.

- The execution time of the checksum function must be reproducible. Hence, Pioneer needs to run in supervisor mode and with interrupts disabled.

The security of Pioneer relies on three important assumptions.

First, the verifier needs to know the exact hardware configuration of the untrusted platform, including the CPU model, clock speed and memory latency, in order to compute the expected untampered execution time. If an adversary can replace or overclock the CPU, he could influence the execution time. Hence, the Pioneer system assumes that the hardware configuration is known by the verification entity and cannot be changed.

Secondly, an adversary could act as a man in the middle and delegate the code to a faster computing device to compute the checksum on his behalf. We call these *proxy attacks*. To avoid this, the Pioneer protocol assumes the existence of an authenticated communication channel between the verification entity and the untrusted execution platform.

Finally, a general problem that remains is the network latency. Therefore, Pioneer assumes the verification entity to be located closely to the untrusted execution platform.

Garay and Huelsbergen [57] also rely on the time execution of a verification agent in their "timed executable agent systems" . Contrary to Pioneer, their system issues a challenge that is an obfuscated executable program potentially computing any function. Hence, the verification agent is mobile, while Pioneer uses a single fixed verification function invoked by a random challenge.

3.6.2 Software Diversity

As indicated in Chapter 1, software diversity tries to thwart the automation phase of a *global attack*. Similar to diversity in fauna and flora, software diversity can strengthen the software community against global attacks. Cohen [12] was one of the first addressing the potential of code diversity. He called it "program evolution" and situated it in the context of operating systems protecting against automated attacks. Collberg *et al.* [36] look at models from biology and history, classify them into primitives, and illustrate how to map them onto a digital world where software also requires defense mechanisms.

Historically, code diversity was used in fault-tolerant systems, e.g. in areas such as aerospace or aviation. Concepts as N-version programming duplicate software and/or hardware to avoid errors affecting overall results. In 1997, Forrest *et al.* [56] motivated why diversity in computer systems is valuable for security as

well. The authors justify their proposal by randomizing the amount of allocated memory on a stack frame to disrupt buffer overflow attacks. Cox *et al.* [45] propose N-variant systems as a mean to protect servers against divergence on servers running multiple instances of the same process. A year later, Anckaert *et al.* [8] propose a self-modifying virtual machine to protect against tampering. In their model they suggest a customized VM that reads customized bytecode. Their system applies diversity on the level of:

- the instruction set architecture;
  - instruction semantics, instruction encoding, and opcode encoding;
  - fetch cycle;
  - program counter;
- the VM code itself.

These three examples illustrate that diversification, whether on the level of a software community, processes, or within a single program, does contribute to the security and protection of the target applications. More results on software diversity as a mean to protect software are bundled in Anckaert's PhD dissertation [6].

3.6.3 Steganography, Watermarking, and Fingerprinting

Over the years, it has become clear that redundancy in executables can be used to embed covert messages [55, 7]. Furthermore, differences between programs with the same functionality can act as a watermark. These differences include: static data, dynamic data (values), but also algorithms, runtime data structures, etc. A *software watermark* is a value or property embedded in the software which proves ownership of the software. In the case, where each software instance has a unique value embedded, identifying the legitimate user of the software, we denote this value as a *software fingerprint*.

Collberg *et al.* [41] sketches the importance of software watermarking as a complement to tamper-proofing and obfuscation. The last decade, many constructions have been made to watermark software: graph-based watermark [40], path-based watermarks [39], and watermarks based on constant encoding [129]. However, it is not that straightforward to embed a robust watermark in software. Typically, the same transformations that embed the watermark, can be used to either distort it, or remove it.

Consider a program P. P_k is the same program P with an embedded secret k, for example a serial number to activate the software for a specific user. Another user has $P_{k'}$, the same program P, but with a different secret k'. Without additional "diversity transformations" attackers can easily compare the two binaries to locate the serial key. This is called a *collusion attack*. Subsequentially, they can:

- steal the serial number;
- crack the software by overwriting the serial verification routine; or
- distort the watermark (if the serial number serves as a watermark).

Collusion attacks are powerful and definitely not limited to word-based comparisons of binary files. Nagarajan *et al.* [99] demonstrate that matching of control flow of program versions can strengthen single-version deobfuscation attacks.

3.6.4 Buffer Overflows and Exploits

Viruses and worms [89, 73, 97] have been a hot topic in the media. Triggered by these virus outbreaks, discussions often mention the choice of operating system. This actually refers to the problem why viruses spread so successfully. One reason could be that the software community is evolving to a monoculture, meaning that most people use the same operation systems and software packages, containing the same type of bugs. This is one of the reasons why viruses, that exploit one or more security holes, are so successful. Buffer overflows [44, 118] are one of the most common exploits. While there exist solutions to solve this type of errors, e.g. bound checking, type safe languages, etc., buffer overflows still exist and attackers find more clever ways to exploit them [114, 27].

Tamper resistance techniques and software diversity, however, can prevent buffer overflows, detect them, or slow down automated attacks. Forrest *et al.* [56] sketches the analogy between diversity in computer systems and diversity in biological systems. Their paper presents some preliminary results on randomizing stack layouts by increasing certain slots with a random times 8 bytes. Such a simple modification could harden a program instance against standardized buffer overflow attacks. Another technique to battle buffer overflow attacks, called *address obfuscation*, is also based on the idea of code diversity [15]. This technique randomizes the code and data sections on the stack by randomizing all the base and start addresses, locations of routines and static data and introducing gaps between objects. While address space layout randomization

(ASLR) nowadays is adopted by major operating systems, some attacks have been proposed [115]. Nevertheless, the same work indicates that ASLR raises the technical bar for attackers performing ordinary buffer overflows.

3.7 Conclusions

In this chapter, we have summarized several state-of-the-art protection techniques that can be integrated in software to protect it against analysis and tampering attacks. The solutions we propose are divided into architectural protection techniques, static and dynamic anti-analysis techniques, and tamper resistance techniques. Although, we presented all these possible techniques separately, it is possible to combine these techniques into one solution. The more protection built into an application, the more the bar is raised for attackers.

Chapter 4

Software Protection against Static Attacks

4.1 Introduction

In this chapter, we propose a method to protect against static control flow analysis. We start by briefly revisiting the concept of static analysis. We explain how reverse engineering is done step by step, followed by a focus on some important techniques to protect against static analysis. In Section 4.4, we explain how our technique strengthens existing solutions. Our protection technique makes it hard for an attacker to statically reconstruct the original control flow graph. Section 4.5 sketches some application models in which we can deploy our technique, while Section 4.6 describes some attacks to investigate its security.

Our main contributions consist of a protection scheme which extends control flow graph flattening such that it is stronger against static control flow analysis, three models that map our scheme onto application scenarios, and some attacks to illustrate the strength of our scheme.

This chapter is based on work described in [24]. The author of this thesis is principal author of this publication.

4.2 Static Analysis

Static analysis is a broad term which basically refers to analyses which do not involve execution of the actual code. Compilers use static analysis techniques to optimize code [5]. Examples include constant propagation and liveness analysis. Reverse engineering is a term which is often used with a malicious connotation. Someone has something and wants to know what it does, how it does this, etc. A reverse engineer typically starts inspecting an object, by disassembling it, and then tries to understand it bit by bit, composing parts, finding patterns, etc. In software, a very similar process happens. First a binary file is disassembled. As a second step, the reverse engineer might opt to decompile the disassembled code into source code. And finally, he will inspect the source code. Note that the reverse engineer can also simply run the code he obtained. However, this is considered a dynamic attack on which we will elaborate in Chapter 5.

4.2.1 Disassembling

When reverse engineering a binary file, a first step consists of disassembling the file into a human understandable format. The disassembly step is the inverse of the assembly stage in compilers. It translates binary code into assembly instructions that conform a particular CPU architecture. While this is a static technique, it is not a perfect technique, as practical disassemblers have to rely on assumptions. A first example is the difference between code and data. Unless something indicates whether a particular byte is code or data, a byte could be either code, data, or both [12, 95]. Furthermore, in many architectures code consists of multi-byte instructions. Hence, in theory instructions could overlap with other instructions and data (although it is seldom the case that a compiler creates such output).

Most disassemblers assume that code and data are not overlapping, and that a program consists of a sequence of non-overlapping instructions. The advantage here for processors is that when an instruction is fetched, incrementation of the instruction pointer marks the beginning of the following instruction. Linn and Debray [84] illustrate that by inserting data in between instructions, so called "junk bytes", the disassembler can be fooled. It will disassemble these bytes as instructions. This technique allows one to disrupt static disassembly. However, in their conclusion the authors mention this is merely 'desynchronizing' the disassembler, as after on average 3 instructions the disassembler 'resynchronizes' with the correct assembly listing.

4.2.2 Decompilation

Decompilation is actually an optional technique that the reverse engineer can apply. If an attacker wants to understand an entire program, he might be confronted with millions of lines of assembly code. A decompiler basically looks for patterns that can be translated into source code. As high-level code is richer and more compact, it is often easier to understand.

4.3 How to Protect against Static Analysis

4.3.1 Code Encryption

While encryption often is presented as the means to protect software statically, it actually shifts the problem, analog to cryptography where secrecy of a message is shifted to secrecy of a key. In an encrypted executable file, original program code is encrypted, and a decryption routine is added to the original program. Hence, code encryption is a form of self-modifying code [91]. Actually, the entire program is treated as data, while the decryption routine remains code. If the latter is easy to analyze, one can 'break' the decryption routine, and decrypt the program. Sequential steps such as disassemblation and decompilation allow to reverse engineer it, as if it were never protected.

Furthermore, not all architectures currently support self-modifying code. Some operating systems enforce a W⊕X policy as a mechanism to make the exploitation of security vulnerabilities more difficult. This means a memory page is either Writable (data) or eXecutable (code), but not both. Encrypted code might as well conflict with virus scanners due to its suspicious behavior (malware also uses self-decrypting code) or due to false-positive signatures matches, i.e. bit patterns that virus scanners scan for. A workaround for this limitation is the use of a virtual machine [82]. If code is compiled just-in-time, the virtual machine can run it. If the virtual machine is secret, and the byte code is encrypted, one has to attack the virtual machine first.

Strictly speaking, one could claim that this is not a real code transformation, as it has nothing in common with syntax rules or program semantics. It rather treats the program as raw data (the cleartext) which is then 'transformed' into ciphertext.

The dynamic attack is much simpler: just let the program run (e.g. in a sandbox), observe where the payload is decrypted in memory, and intercept it

(e.g. perform a memory dump). In Chapter 5 we present a technique to limit program exposure to memory dumps.

4.3.2 Code Obfuscation

Data flow Obfuscation

Data obfuscation or data flow obfuscation transforms data structures and consequently data flow in programs [37]. Data flow transformations are hard and costly. Even compilers sometimes work with heuristics [5].

The results in this chapter aim to protect control flow. However, we extend a technique called *control flow graph flattening*, which in essence translates a control flow problem into a data flow problem. As such, our technique focuses on data, namely the protection of statically embedded control flow data.

Control Flow Obfuscation

"Spaghetti code" is often mentioned in the context of obfuscation. Often, `goto` statements are undesired as they can lead to unstructured code: i.e. code where control is transferred from here to there, instead of using defined high-level structures such as loops, branches, etc. Badly written or very unstructured code might indeed challenge both human and machine inspection. However, it is very hard to prove certain properties or security assertions of unstructured code.

Control flow graph flattening is a more structured control flow transformation. Wang *et al.* [138, 137] call it "program degeneration" as it transforms a control flow graph into an abstract, degenerated form. The rest of this chapter presents a scheme which extends the strength of control flow graph flattening against static analysis attacks.

4.4 A General Model for Hiding Control Flow

This section proposes a general model for hiding control flow relations in C-like programs to protect them against static control flow analysis. First, we explain what control flow graph flattening is and illustrate why it is successful as protection against static control flow analysis. Secondly, we propose a scheme, complementary to control flow graph flattening, which does not leak any control

flow graph information statically. As a result, together with control flow graph flattening our technique makes it hard to statically reconstruct the original static control flow graph of a program.

Furthermore, instead of relying on ad hoc security by using variable aliasing and global pointers to complicate data flow analysis in a flattened control flow graph [79], we try to base our security claims more on information theory, data flow, and cryptography. Our formal model allows to specify which components to hide in the program (e.g. a secret value or function) such that no control flow information is leaked.

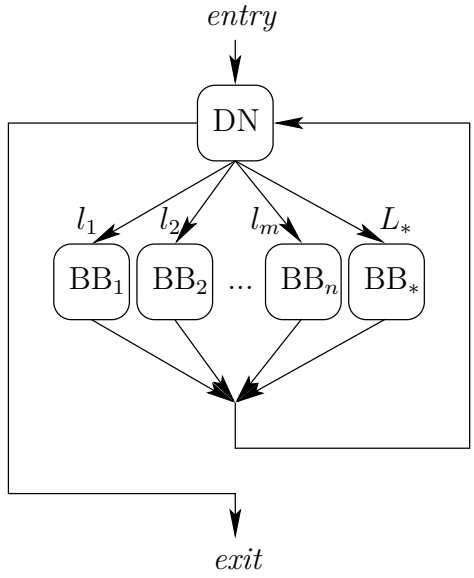
4.4.1 Control Flow Graph Flattening

The Control Flow Graph

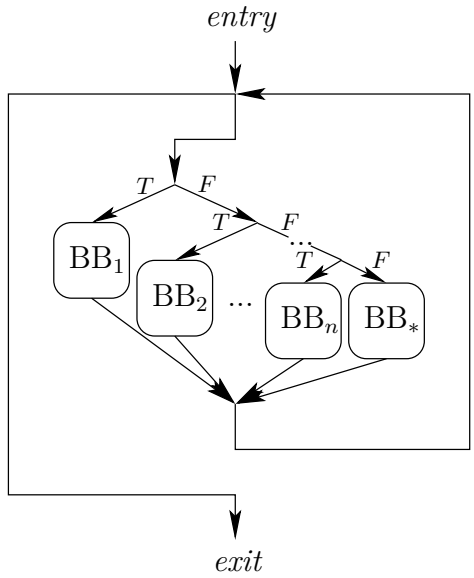
Consider a program P as a graph $G = \langle V, E \rangle$, where V represents the set of basic blocks $\mathrm{BB}_1, \mathrm{BB}_2, \ldots, \mathrm{BB}_n$ and E represents the set of control flow edges $e_1, e_2, \ldots, e_m$. This is usually called the control flow graph (CFG) of P. Conventional programs usually contain hard-coded information denoting how and where control can be transferred from one to another block. If we collect all this information and represent it graphically we get a static control flow graph, where nodes represent basic blocks and directed edges represent control transfers. If we would trace a program dynamically, we can reconstruct a part of this graph (i.e. the executed path), built out of a subset of V and a subset of E.

CFG Flattening

In a flattened version of a program, a dispatcher node DN gives control to a specific basic block BB_i out of n basic blocks [137]. After executing a block, control is transferred back to the dispatcher node DN, which decides how to continue. At every iteration, any block is a possible candidate to be selected next. Because the dispatcher node makes decisions based on a switch variable, the control flow problem has become a data flow problem. At first glance, it is usually not clear in which order the basic blocks will be executed (unless the dispatcher node is easy to analyze and the control flow information is still hard-coded in the basic blocks). Whereas in a traditional program most basic blocks can give control to 1 or 2 other basic blocks (2 in the case of a decision point), control in a 'flattened' program can be transferred to n other points, n being the number of basic blocks. Not performing data flow analysis (e.g.



(a)



(b)

Figure 4.1: (a) A flattened control flow graph. (b) An `if-else if-else if-`... structure, which is equivalent to the flattened program in (a).

constant propagation) would yield in $O(n^k)$ possible execution paths, where k denotes the number of executed blocks.

Figure 4.1(a) shows a flattened CFG implemented as a switch structure, nested in a loop. DN represents the dispatcher node, BB_i the basic blocks, and l_i the labels. The dispatcher node decides which basic block to execute based on which label value is assigned to the switch variable. If a label l_i is not present, control will be transferred back to the dispatcher node DN. This is achieved via a sort of fall-through path. Consider $L = \{l_1, l_2, \ldots, l_m\} \subset \{0,1\}^n$ the set n-bit labels in a program, then L_* matches all remaining labels, namely $\{0,1\}^n \backslash L$. The basic block BB_* is a special basic block that will be executed each time the switch falls through because no match to the value of the switch variable is found in L. In C, it is possible to specify a basic block in this fall-through path, using the `default` label. Figure 4.1(b) shows a nested `if-else` CFG which offers functionality equivalent to the flattened CFG. [1]

4.4.2 Strengthening CFG Flattening

In this section we gradually present our protection scheme. Step by step we perform improvements on a flattened control flow graph to make analysis harder.

Setup Phase

To illustrate our method of transforming a program in a flattened one, which is hard to analyze statically, we start with the implementation as published by László and Kiss [80]. In this model a single variable, the "switch variable" (`swVar` in the listing below), contains a label value used to switch control to the corresponding basic block. In each basic block new values are assigned to this variable. We use the following example from their paper:

```
case 2:
  if (c <= 100)
    swVar = 3;
  else
    swVar = 0;
```

First, note that constants are assigned, which should be hard-coded labels if no fall-through path is present (i.e. the `default` label in C). Secondly, the

[1] We assume that a basic block does not "fall through" to the next basic block, which would be the case if one omits the `break` statement in C's `switch` structure.

branch still contains an `if-else` construct. Hence, the code still reveals where execution will branch. The assignments in this example are of the form $x = l_i$ where x represents the switch variable and l_i denotes the label value of the next basic block to execute. This allows an attacker to perform local analysis, which we define as follows:

Definition 1 *A **local analysis** involves analysis of a small portion of code, located in or around a particular place.*

Within our context, local analysis means the analysis of a single basic block, i.e. its contents, its label or address, etc. However, we only focus on static information in this chapter. To deduce the next block in the example above, one just has to look up the block appointed by label l_i. In order to find all possible preceding blocks, an attacker has to iterate over all basic blocks and scan for assignments of the form $x = l_i$.

In the following sections we propose steps to increase security of this control flow graph flattening algorithm. Our goal is to force the attacker to do a global analysis, even if he wants to perform a local attack.

Step 1: Remove Hard-Coded Control Flow Values

As the implementation proposed by László and Kiss is susceptible to local analysis attacks, we promote the use of relative variable updates. Each assignment to the switch variable is a relative update of its current value. Hence, the example could look like this:

```
case 2:
  if (c <= 100)
    swVar = swVar + 1;
  else
    swVar = swVar - 2;
```

As such, control flow information is less explicit. Assignments of the form $x = x + \alpha$ yield that possible targets (successor blocks) can only be derived if the actual value of x is locally available. If not the case, the attacker should perform backward analysis to compute predecessor blocks, and for each of these again predecessor blocks, etc. This yields that a simple local attack requires a global analysis with worst case complexity $O(n^k)$ with k the number of preceding blocks, leading to the particular block, and n the number of basic blocks.

Step 2: Use a Single Uniform Statement

As mentioned earlier, several basic blocks in László and Kiss's implementation still contain branches: the `if-else` reveals a decision point. To replace these hard-coded branches, we propose the use of a single uniform statement. A branch namely depends on a condition which always returns *true* or *false*. With the appropriate logic we can map *true*/*false* values onto 1/0. For simplicity, we assume that conditional expressions result in 0 or 1. These values can then be used in an expression. The example above could be reduced to:

```
case 2:
  swVar = swVar + 1 - (c <= 100) * 3;
```

We get one uniform statement, which can be represented as a function. We will call this function the *branch function* $B()$:

$$B(x) = x + \alpha + y \times \beta$$

Here x denotes the switch variable, and y denotes the condition. Both are represented by n-bit variables, but due to the outcome of the condition y is always of the form $00\ldots0a$ where a is the Boolean result 0/1 of the conditional expression. '+' and '×' just represent addition and multiplication. Furthermore, we notice two hard-coded constants $\alpha, \beta \in \{0,1\}^n$. α and β can always be chosen such that for any basic block BB_s and any two target blocks BB_{t_1} and BB_{t_2}, respectively with the labels l_s, l_{t_1}, and l_{t_2}:

$$\forall\ l_s, l_{t_1}, l_{t_2}, \exists\ \alpha, \beta\ :\quad l_{t_1} = l_s + \alpha \text{ and } l_{t_2} = l_s + \alpha + \beta$$

Jump statements, such as C's `goto`, `break`, or `continue`, only transfer control to 1 successor block. Hence, they have a simpler form: $B(x) = x + \alpha$. To increase confusion, one could add opaque predicates [42], i.e. conditional expressions which are known at compile-time but which are hard to compute statically. Once the predicate is added, we get a similar form as above. In this setup, we only consider always-*true* and always-*false* predicates whom outcome we will map onto $\{0,1\}$.

Step 3: Use Non-Local Values

While the previous step seems to be an improvement in terms of *def-use* chains (the switch variable is always defined in the previously executed block, which

is not known at first glance in a flattened CFG), local analysis is still trivial. Namely, the value contained by the switch variable is always one of the labels leading to the basic block containing this code. Hence, it is locally available. To solve this, we propose to use an earlier computed value. For example, consider the label of the previous block. During a static attack the adversary does not know which block preceded the other block, unless he solves the statements in each basic block, which themselves depend on other values from other blocks, etc. Instead of $x_{i+1} = x_i + \alpha + y \times \beta$ where x_i is locally available, we would get $x_{i+1} = x_{i-1} + \alpha + y \times \beta$.

However, if x_{i-1} in block BB_a contains the label of the previously executed block, we might have multiple possible label values if several blocks can give control to BB_a. If BB_a in turn gives control to another block BB_b, we cannot just rely on a single value contained by x_{i-1}. To solve this we need to:

- create two labels for BB_b (two paths towards it), but this only propagates the problem;
- introduce a corrector value; or
- use a dummy basic block where all predecessor blocks jump to and use the label of that block.

We opt for the last solution. The secret value x_{i-1} in BB_b is in this case a single, unique value computed at runtime in a dummy basic block and identical for all paths leading to the dummy block. The dummy block transfers control to BB_b without passing information from its predecessor blocks. We propose to use a runtime value which is not hard-coded, hence not susceptible to static analysis. For this, the fall-through path of a switch becomes very handy. In this basic block we could specify how to update the switch variable and store its current (runtime) value for use in the next basic block. As a consequence, two or more blocks can (eventually) give control to a common successor block, by first giving control to the same dummy basic block(s). This dummy block will just cause the switch loop to iterate via the "fall through path" and transfer control to the target block BB_b.

Step 4: Make Values Hard to Compute

While it is already hard in a flattened control flow graph to analyze what a predecessor block was, we can even make it computationally hard for the attacker to retrieve x_{i-1} out of x_i, the label of the executing basic block. For this, we could for example use a one-way function which makes it computationally hard

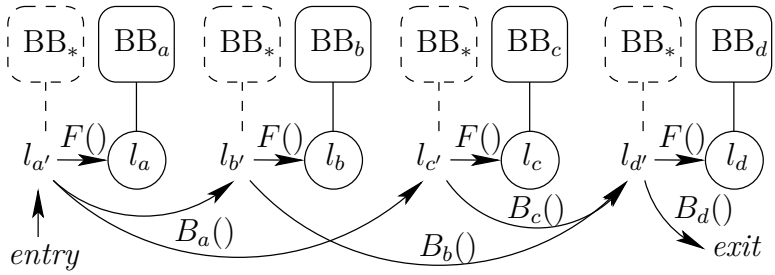


Figure 4.2: How control flow data flows through transition function $F()$ and four branch functions: $B_a(), B_b(), B_c()$, and $B_d()$.

for the attacker to compute the preimage of a label statically. Consequently, at construction time the defender has to precompute these hash pairs himself.

From now on, we will denote our one-way function as the *transition function* $F()$. For each control transfer, a label value flows through this function. For the security of our overall scheme we propose the following properties for $F()$:

One-way. This property computationally withholds the attacker from doing backward analysis. Namely, to get the previous value of the switch variable in a *use* site, one has to do backward liveness analysis to locate the *def* site.

Bijective. In a bijection, each unique label will be mapped onto another unique label. The importance of this property will become clear in Section 4.6.1.

Diffusion. Preferably, $F()$ has good diffusion properties such that constant or range propagation [19] becomes significantly harder.

Examples of one-way functions, and thus good candidates for our transition function, include:

The discrete logarithm. The discrete logarithm over a finite field $\mathrm{GF}(p)$. In $\mathrm{GF}(p)$ we can use $F(x) = g^x \bmod p$ with g generator of the field and p a large prime. It is hard to compute x for a given $F(x)$. Furthermore, this function is bijective. If we also specify our branching function in $\mathrm{GF}(p)$ we get: $B(x) = (x + \alpha + y \times \beta) \bmod p$ with constants $\alpha, \beta \in \mathrm{GF}(p)$ and $y \in \{0, 1\} \subset \mathrm{GF}(p)$. In this case $B()$ is also a bijection, and invertible. The latter property of $B()$ is required to compute α and β.

Cryptographic hash functions. Hash functions in cryptography are typically used as compression functions from m to n bits. However, we can

also use them for n-to-n bit conversions. Cryptographic hash functions are crafted in such way that they are fast, but still hard to invert. Strictly speaking cryptographic hash functions have collisions, and thus they are not bijective. However, finding collisions is considered to be hard, such that for the purpose within our protection scheme we can consider them as a good candidate. The proof in Section 4.6.1 elaborates on this. Defining our transition function would yield: $F : \{0,1\}^n \rightarrow \{0,1\}^n : F(x) = hash(x)$, while a bijective branching function could be: $B(x) = x \oplus \alpha \oplus y \times \beta$.

We will implicitly consider the second example, where $F()$ is a cryptographic hash function, and where $B(x)$ uses exclusive OR as binary operator. Section 4.6.3 reasons on why it is so important to choose a binary operator with good properties. We will also use the term 'preimage', which is the inverse image of a hash value $y = F(x)$: $F^{-1}(y) = x$. Finding preimages for hash functions is a hard problem and one of the basic properties of this cryptographic primitive.

Overall Structure

To summarize, our scheme looks as follows:

- the fall-through basic block BB_* contains:
  - a new variable z storing the runtime value of the switch variable (which lead to BB_* and which is not hard-coded in the code): $z = x$, and
  - a call to our transition function: $x = F(z)$.
- all other basic blocks BB_i contain:
  - an instantiation of $B()$ with precomputed α and β. This $B_i()$ operates on the "runtime label" stored in z, which is from a static point of view a secret value: $x = z + \alpha_i + y_i \times \beta_i$. Remember that the label value in z is the preimage of the label that is used to transfer control to BB_i.

The transition function $F()$ is used for its one-way and diffusion properties. At runtime, it will allow a dynamic label to "fall through" to a static label. The branch function $B()$ is used to transfer control to other blocks. This as well is achieved via a dummy block, linked to by the dynamic label.

Finally, we get a structure that looks like the example in Figure 4.3(b). Figure 4.3(a) illustrates the CFG of a simple `if-else` branch. Here control can

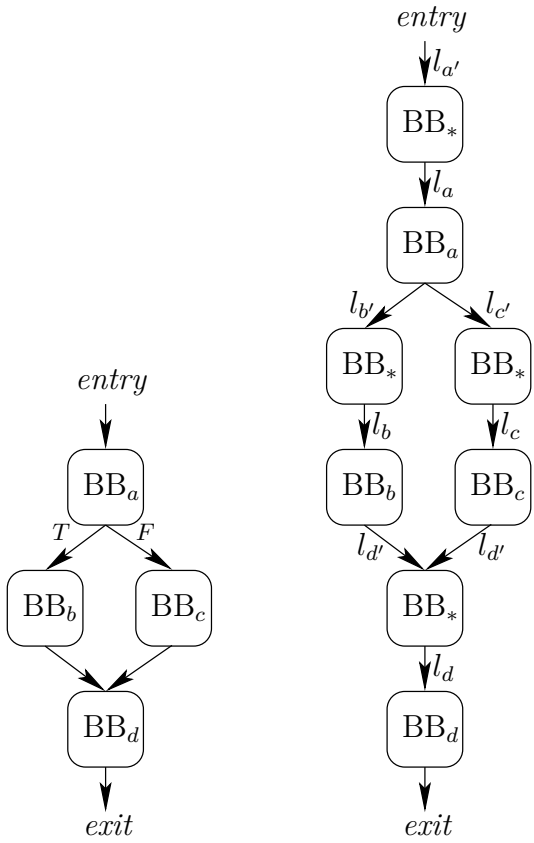


Figure 4.3: (a) A simple `if-else` branch CFG. (b) The transformed "graph" according to our overall protection scheme.

branch from BB_a to BB_b or BB_c. The transformed control flow graph (after removing the flattening, and omitting passes through the distribution node DN) is illustrated in Figure 4.3(b). Labels with a prime index are preimages (and thus runtime values) of hard-coded labels, e.g. $l_{a'}$ is the preimage of l_a. The BB_* blocks all represent the same block, namely the block that matches all label values that are not hard-coded in the switch. Figure 4.2 illustrates how these labels (i.e. the control flow data) flows through the transition function $F()$ and branch functions $B_i()$. In this figure, the labels in a circle are hard-coded in the program, while the other labels represent runtime values.

When applying such a protection scheme on a program's control flow, the defender can choose all labels at will and compute the required hard-coded constants, such that relations in the flattened graph still represent relations of the original control flow graph. Because $F^{-1}()$ is hard to compute, the defender has to generate a set of small hash chains in advance. From this set he then picks his labels and their preimages. If values are chosen randomly, attackers cannot exploit any patterns.

So far our scheme forces an adversary to perform global analysis if he wants to investigate local control flow. In Section 4.5 we sketch scenarios in which we can hide a program's control flow (and other internals relying on it) entirely, such that even global analysis techniques cannot be applied.

4.5 Application Models

In this section we describe application scenarios of our protection scheme. To hide control flow statically, we just apply our scheme of Section 4.4.2 and either split off a portion of the program (data or code), or we rely on a statically hard problem. First, a program can be split into a program and a key. Without the secret key, one can derive nor the first basic block, neither the subsequent blocks. In a second scenario, the program splits off a small function which can be protected in hardware, such as a dongle or a smart card. And last, we present a "stand-alone" solution that computes a key at runtime, but still makes it hard to derive the key statically (to eventually prevent static control flow analysis).

In the first two application scenarios we propose the attacker cannot run the code without an additional secret component. Random guessing of the switch variable should be hard, as will be explained in Section 4.6.1. Constructing the secret transition function is not an option either as the attacker has no input-output pairs to $F()$. Only in the third scenario, all code is stored on the untrusted host. However, due to the nature of opaque predicates [42], the

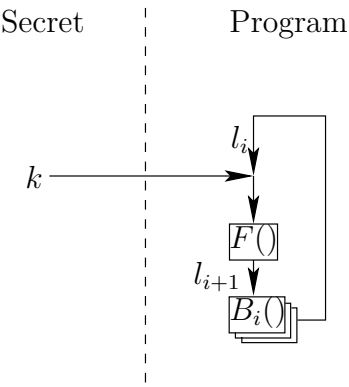


Figure 4.4: A scenario with just a single secret value, the *key*.

attacker cannot prove that the generated label values (and thus the execution as well) will always be the same.

4.5.1 A Secret Key

A first example hides a program's secrets behind a secret key. This idea is used for centuries as the main primitive in cryptography, where message confidentiality is protected by a (usually smaller) secret key. Once we succeed in hiding a program's control flow, we can make other analyses flow-sensitive to reduce their efficiency.

The advantage of this key scenario is that we can distribute applications in advance of sending the key. However, if we rely on secrecy of a key, we cannot store it on an untrusted host. Once the key is revealed, one can perform static techniques such as constant propagation. Another property of our scheme is that it allows to merge programs (each key k_i will yield execution of a program P_i). These programs can even have basic blocks in common such that their actual CFGs will be interweaved.

In our protection scheme, we force the attacker to do global analysis, starting at the entry point of the CFG. Consider the label of the first basic block l_1, then the secret key k can be a preimage value of l_1: $k = F^{-1}(l_1)$.

A practical scenario could be remote voting applications on PCs as this concerns programs that execute once (after installation). The scenario could look as follows: one big program is installed in advance, at moment t_1 a key k_1 is sent

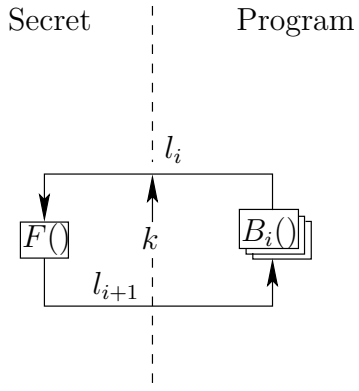


Figure 4.5: A scenario with a single *function* implemented in hardware or ran remotely on a server.

to execute functionality $f_1()$, at moment t_2 another key k_2 for $f_2()$, etc. Keys could be sent via the network from one or more servers.

4.5.2 A Secret Function

Our scheme uses two types of functions: the branching functions $B_i()$ and the transition function $F()$. Basically, the branching functions contain all the control flow information, needed for branching and jumping. The transition function, however, is needed to make our scheme stronger (making data flow analysis one-way).

In this scenario, we rely on a "trusted party" that has computational abilities. As such, it can compute secret function $F()$ in a challenge-response alike way. This trusted party could be a server, but also a smart card, or a hardware dongle. Without this additional piece, the software cannot run and attackers cannot learn from it. Figure 4.5 illustrates this model.

4.5.3 Stand-Alone Solution

Apart from the applicability of the models above, we also propose a "stand-alone" application where control flow is protected by a secret value which is computed at runtime. With the existence of opaque predicates [42], we can construct a 'random' secret n-bit value k by using one of the following constructions:

$$k = p_1 \parallel p_2 \parallel \cdots \parallel p_n$$

where all p_i represent always-*true* or always-*false* opaque predicates which map onto 1 or 0, and $\parallel$ denotes the concatenation of bits. The disadvantage of this combination is the bitwise combination. If one or more predicates are broken, bits are revealed. In our scheme however, this value is first processed by $F()$ which has good diffusion properties, and thus reduces the risk of being exploitable. An alternative construction is:

$$\begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ r_{21} & r_{22} & \dots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & \dots & r_{nn} \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}$$

or $K = RP$ with R a hard-coded $n \times n$ random matrix of full rank over GF(2) and K and P matrices with dimension $n \times 1$ where P is the vector composed of all p_i bits, which come from independent opaque predicates. To a static attacker this should have the same probability as a random bit. Dodis *et al.* use a similar technique to 'blend' two sources of randomness [54].

It should be clear that this scenario makes static analysis hard, but does not prevent an attacker from just running the code. He even might repeatedly run the code with different inputs and bypass the predicates as he always observes generation of the same key.

4.6 Attacks

4.6.1 Brute Force Attacks

Even though in static analysis the inputs are not known, several global analyses succeed in extracting information. Techniques include: constant propagation, range propagation [19], etc. Also, techniques such as abstract interpretation have been proven useful, e.g. for breaking opaque predicates [47].

We will prove that much of the strength of these techniques does not apply to our models. First, consider the model where $F()$ is secret. As $F()$ is unknown, the attacker cannot propagate constants (or ranges). Secondly, consider our model where we either keep a key k secret, or construct it at runtime using opaque predicates. If the n-bit key k is uniformly distributed, its entropy $H(k)$

equals n bits. We will show that brute-forcing any of the intermediate values of the switch variable, has the same chance of succeeding as long as the switch variable's entropy is preserved during execution.

Proof 1 *Consider the model in GF(2^n), where $F(x) = hash(x)$ is a strong hash function, and $B(x) = x \oplus \alpha \oplus y \times \beta$.*

Remember that $y \in \{0, 1\}$ as y represents the mapped outcome of a condition (or predicate). With probability $P(y = 0)$ we get $B(x) = x \oplus \alpha$, while with probability $P(y = 1)$ we get $B(x) = x \oplus \alpha \oplus \beta$ with α, β constants. As exclusive OR has a balanced outcome, it is not loosing information. Consequently, $x \rightarrow B(x)$ is a bijection. It maps GF(2^n) onto GF(2^n). Furthermore, bijections preserve entropy because probabilities are just mapped onto other values. Hence, we conclude: $B(x)$ preserves entropy of the switch variable during program execution.

For the hash function, it is somewhat different. Strictly speaking cryptographic hash functions are not fully bijective. However, finding collisions for good hash functions is a hard problem (computationally expensive). As such, we can conclude that our attacker cannot exploit these collisions. We consider therefore $hash(x)$ also bijective for the inputs we use. And thus it also preserves entropy.

Note: for GF(p) we can provide a similar proof. It is more sound though, as $B(x) = (x + \alpha + y \times \beta) \bmod p$ is a full bijection, as well as $F(x) = 2^x \bmod p$.

4.6.2 Attacking the Transition Function

When the transition function $F()$ is known, one can generate the hash chains offline. For n bits this implies 2^n values. Using these chains one can search for preimages of a certain (hard-coded) label l_i, allowing to solve the branch function equation based on the –unknown– preimage of this label.

However, computing the entire hash chain for $F()$ becomes impractical for large numbers of n. Consider 2^{10} hard-coded 32-bit labels. This yields the probability of finding a preimage of one label $2^{10}/2^{32} = 1/2^{22}$. First, due to the one-way properties of $F()$ the attacker would only be able to propagate this value forward, only reconstructing a part of the static control flow graph. Secondly, for cryptographic hash functions an output of 160 bits is recommended. Thus, even if there were millions of basic blocks, an attack would still require 2^{140} attempts to find a usable preimage value.

4.6.3 Attacking the Branch Function

If $B()$ is a weak construction, our scheme becomes vulnerable to attacks as well. First, consider the use of bitwise ANDs or ORs in $B()$ instead of the exclusive OR. Then $B()$ would lose information as AND and OR are lossy. On average $x \rightarrow x \,\&\, \alpha$ (with α constant, and '&' bitwise AND), loses 50% of information (i.e. half of the bits are cleared due to zeros in α). Applying this repeatedly would yield very low entropy, allowing brute force attacks.

Secondly, while bitwise binary operators are fast, they propagate differences. If an attacker detects biases, he can propagate those through $B()$. Nevertheless, we still have $F()$, which in case of a one-way function should have good diffusion properties. Without $F()$, our scheme is vulnerable to an attack exploiting differences. Consider the implementation as is in Step 2 (see Section 4.4.2), neglecting the fact that label values of an executing block are visible in the case label. Consider the statement:

```
swVar = swVar + 1 + (cond) * 3;
```

Here we can deduce that the difference between target label l_1 and l_2 is 3. Consider a trivial flattened program with labels $L = \{1, 2, 7, 10\}$. The only possible target labels are label '7' and label '10'. In a real program, with m labels we would have $O(m^2)$ possible differences. It is obvious that the more basic blocks (and thus labels) a flattened program has, the more complicated this difference matching becomes. From Step 4 on, our scheme is no longer vulnerable to this difference attack on labels due to the fact that we no longer jump to hard-coded labels, but to their preimages. These values are not hard-coded, but they are matched by the fall-through label L_*.

A third attack is caused by converging control flow paths. Consider the branch in Figure 4.3(b). As both BB_b and BB_c give control to BB_d, one can deduce that their branch functions map onto the same preimage. Consequently, we get the following equation:

$$F(l_{b'} \oplus \alpha_b) = F(l_{c'} \oplus \alpha_c) \tag{4.1}$$

which yields: $l_{b'} \oplus \alpha_b = l_{c'} \oplus \alpha_c$. Furthermore, the branch function in BB_a reveals that $\beta_a = l_{b'} \oplus l_{c'}$. Solving this set of equations yields $\beta_a = \alpha_b \oplus \alpha_c$. This difference can be exploited in a similar way as the attack on the reduced scheme above. The complexity is equivalent to the one of the example above: $O(m^2)$ for m hard-coded α values. Several solutions exist:

Split basic blocks. Splitting basic blocks increases the amount of α values and β values. Hence, the complexity will also grow in the amount of basic blocks in both paths.

Opaque predicates. As mentioned earlier, opaque predicates [42] allow us to transform each branch function $B_i()$ to the form $B(x) = x \oplus \text{cst}_1 \oplus (p) \times \text{cst}_2$ with p an opaque expression. If the attacker cannot distinguish p from a real expression, he will assume cst_2 to be β, and cst_1 α while this is not the case. In case of an always-true predicate α can be split into two components, while in case of an always-false predicate cst_2 will present a fake β. This also increases the matching complexity.

We propose a stronger solution. Let the branch function in BB_b map onto $l_{d'}$, while the one in BB_c maps onto $l_{d''} = F^{-1}(l_{d'})$. Hence, a target label value is never used twice. Instead, in case of converging control flow, one path jumps to the first preimage and another to the second preimage of that target label value. In this case, it is no longer possible to eliminate $F()$ as was the case in equation (4.1). Instead, we get:

$$l_{b'} \oplus \alpha_b = F(l_{c'} \oplus \alpha_c)$$

4.6.4 Other Attacks

While all previous attacks focus on exploiting our protection scheme, we want to indicate for completeness that adversaries also can use other information to guess the execution order of basic blocks. Examples include: functions (`fopen()` has to be called before `fclose()`), names and liveness of variables, etc. Hence it is always better to apply this technique in combination with other techniques such as variable renaming, dereferencing of variables, function pointers, and other obfuscation techniques.

4.7 Conclusions

This chapter proposed a method to securely embed control flow information in a program, without statically leaking control flow information. Our solution is based on control flow graph flattening. It extends work of other researchers in a structured and strong way such that it becomes hard for attackers to statically reconstruct the original control flow graph.

Our first result forces an adversary to perform global analysis to understand local control transfers. This result holds for both forward and backward analysis. Secondly, we propose three application scenarios which allow to hide control flow behind a secret. This includes a secret key or secret function (in software or hardware), or a construction based on the combination of opaque predicates. To illustrate the strength of our scheme in these scenarios we explained the resistance to several attacks.

If an attacker is unable to gather control flow information, other internals of the program can be made flow-sensitive and thus more resistant to static analysis, e.g. by using aliasing, function pointers, etc. Therefore, we consider our technique a contribution in the field of static analysis protection.

Chapter 5

Software Protection against Dynamic Attacks

5.1 Introduction

As mentioned in Chapter 3, one of the main concerns for software providers is protecting their software from reverse engineering. If a competitor succeeds in extracting and reusing a proprietary algorithm, the consequences may be significant. Furthermore, secret keys, confidential data, or security related code are often not intended to be analyzed, extracted, stolen, or corrupted. Even in the presence of legal safeguards such as patenting and cyber crime laws, reverse engineering remains a considerable threat to software developers and security experts.

In many cases, the software is not only reverse engineered, but also tampered with, as exemplified by the proliferation of cracks for gaming software and DRM systems. In a branch jamming attack, an attacker replaces a conditional jump by an unconditional one, forcing a specific branch to be taken even when it is not supposed to under the foreseen conditions. Such attacks could have a major impact on applications such as licensing, DRM, billing, and voting.

Before changing the code in a meaningful way, one always needs to understand the internals of a program. If one would change a program at random places, one could no longer guarantee the correct functioning of the application after modification. To avoid static code inspection, code is often obfuscated or encrypted. The latter, however, loses its strength at runtime as code is decrypted

prior to execution. In parallel to analysis protection, several papers present the idea of *self-checking code* [26, 64] that is able to dynamically detect changes to critical code. These schemes, unfortunately, do not protect against code analysis.

In this chapter, we propose a technique that aims to thwart both analysis and tampering attacks by means of encryption. Many suggest to apply encryption in the domain of software protection against malicious hosts. However, little information seems to be available on the implementation aspects or cost of the different schemes. This chapter tries to fill this gap by presenting our experience with several encryption techniques. Furthermore, as soon as code is decrypted in memory, it becomes vulnerable again to memory dump attacks. Our main contributions consist of a new type of software guards that limits the time that code is exposed in memory. For this, we rely on-demand encryption. In addition, we improve tamper resistance by using a scheme based on the placement of these guards in a function call graph. Finally, we present performance results of our technique, and a heuristic to improve the performance-versus-security ratio. This chapter is based on [23] and [25]. The author of this thesis is principal author of both publications.

In Section 5.2 we explain the pros and cons of bulk encryption versus on-demand encryption. We propose an on-demand encryption scheme in Section 5.3 of which we present performance results in Section 5.4. Additionally, we define a heuristic to trade off security versus performance. Before drawing conclusions we list several attacks and improvements to our scheme in Section 5.5.

5.2 Code Encryption

The goal of encryption is to hide information. Originally, it was applied within the context of communication, but has become a technique to secure a broad range of critical data, either for short-term transmission or long-term storage. More recently, commercial tools for software protection have become available. These tools need to defend against attackers who are able to execute the software on an open architecture and thus, albeit indirectly, have access to all the information required for execution.

This section provides an overview of runtime code decryption and encryption. One can also refer to this as a specific form of self-modifying or self-generating code [91]. Encryption ensures the confidentiality of data. In the context of binary code, this technique mainly protects against static analysis and tampering. For example, encryption techniques are used by polymorphic viruses and polymorphic shell code. In this way, they are able to bypass intrusion

detection systems, virus scanners, and other pattern-matching interception tools.

5.2.1 Bulk Decryption

We refer to the technique of decrypting the entire program at once as *bulk decryption*. The decryption routine is usually added to the encrypted body and set as the entry point of the program. At run time this routine decrypts the body and then transfers control to it. The decrypting routine can either consult an embedded key or fetch one dynamically (e.g. from user input or from the operating system). The main advantage of such a mechanism is that as long as the program is encrypted, its internals are hidden and therefore protected against static analysis.

Another advantage is that the encrypted body makes it hard for an attacker to statically change bits in a meaningful way. Changing a single bit will result in one or more bit flips in the decrypted code (depending on the error propagation of the encryption scheme) and thus one or more modified instructions, which may lead to program crashes or other unintended behavior due to the brittleness of binary code.

However, as all code is decrypted simultaneously, an attacker can simply wait for the decryption to occur before dumping the process image to disk.

5.2.2 On-Demand Decryption

In contrast to bulk decryption, where the entire program is decrypted at once, one could increase granularity and decrypt small parts at the point in time when they are actually needed. Once they are no longer needed, they optionally can be re-encrypted. This technique is for example applied by Shiva [93], a binary encryptor that uses obfuscation, anti-debugging techniques, and multi-layer encryption to protect x86 binaries using the ELF format [130].

On-demand decryption overcomes the weaknesses of revealing all code in the clear at once as it offers the possibility to decrypt only the necessary parts, instead of the whole body. The disadvantage is an increase in overhead due to multiple calls to the decryption and encryption routines.

5.3 On-Demand Decryption Framework

In this section, we introduce our on-demand decryption scheme. The granularity of this scheme is the function level, meaning that it decrypts and encrypts an entire function at a time.

5.3.1 Principle

The scheme relies on two separate techniques, namely integrity checking and encryption. The techniques from integrity checking are used to compute the keys for decryption and encryption. The integrity checking function can be a checksum function or a cryptographic hash function. Essentially, it has to map a vector of bytes, code in this case, to a fixed-length value, in such a way that it is hard to produce a second byte vector resulting in the same hash.

The idea is to apply the integrity checking function $h()$ to (the code of) a function f_a to obtain the key for the decryption of another function f_b. Using the notation of $D_k()$ for decryption and $E_k()$ for encryption with key k, we have:

$$f_b = D_{h(f_a)}(E_{h(f_a)}(f_b))$$

To this end, f_b needs to have been encrypted with the correct key on beforehand (i.e. $E_{h(f_a)}(f_b)$, denoted by $\bar{f_b}$). We call the code responsible for the on-the-fly decryption of $\bar{f_b}$, including the computation of the decryption key $h(f_a)$, a *crypto guard*. Hence, we get the following definition:

Definition 2 *A **crypto guard** is code responsible for decryption, respectively encryption, of a portion of code. This code includes the computation of a key by hashing another portion of code.*

In our scheme, crypto guards work on the function level. They on-the-fly decrypt and encrypt functions. In order to make program tampering hard, we would like our guard to have at least the following properties:

- if one bit is modified in f_a, then 1 or more bits in f_b should change after decryption; and
- if one bit is modified in $\bar{f_b}$, then 1 or more bits should change in f_b after decryption.

For the first requirement, a cryptographic function with f_a as key could be used. For example, Viega *et al.* [134] use the stream cipher RC4 where the code of another function is used as the key. The advantage of an additive stream cipher is that encryption and decryption are the same computation, thus the same code. It is also possible to construct stream ciphers out of block ciphers (e.g. AES [46]) using a suitable mode of operation (e.g. CTR or OFB). The disadvantage of stream ciphers is that bit flipping in ciphertext results in bit flips in the plaintext. For more information on stream ciphers, block ciphers, and their modes of operation, we refer to [94]. The major disadvantage of these ciphers is that they are relatively slow for our case, and relatively large as well. Furthermore, note that we still require the integrity-checking function that also serves as a one-way compression function, because each cipher requires a fixed-sized key.

From a cryptographic point of view we require a second image resistant hash function and a strong cipher in a secure encryption mode with suitable error propagation properties (e.g. PCBC). However, size and speed of these algorithms is essential for the overall performance of the protection scheme as its security assumes inlining the guard code. This results in more code to be hashed and decrypted, and thus a higher cost. It is also possible to link the hashing itself to other code by using a keyed hash function, such as HMAC.

As software tamper resistance typically consists of techniques that make tampering with code harder, we can demonstrate that our crypto guards offer protection against tampering. Namely, using code of f_a to derive a decryption key, could be seen as a self-checking technique. Using this decryption key to decrypt code of f_b, denoted $\bar{f}_b$, represents the implicit checking mechanism. Hence, modifying f_a will result in an incorrect hash value (i.e. decryption key), and consequently incorrect decryption of $\bar{f}_b$.

Furthermore, changing $\bar{f}_b$ will result in one or more changes to f_b. In case of an additive stream cipher a bit change in the ciphertext will correspond to a bit change the plaintext at the same location. However, if this plaintext itself is used as key material at a later stage, this will result in incorrect code decryption because another decryption key will rely on it. Furthermore, due to the brittleness of binary code, a single bit flip might change the opcode of an instruction, resulting in an incorrect instruction to be executed, but also in desynchronizing the next instructions [84], which most likely will lead to a crashing program.

Another advantage of this scheme is that the key is computed at run time, which means the key is not hard-coded in the binary and therefore hard to find through static analysis (e.g. entropy scanning [116]). The main disadvantage is performance: loading a fixed-length cryptographic key is usually more compact

and faster than computing one at run time, which in our case involves computing a hash value.

Although cryptographic hash functions and ciphers are more secure, we use a simple XOR-based scheme – which satisfies our two properties – to minimize the performance cost in speed and size after embedding the crypto guards. We therefore do not claim that our encrypted code is cryptographically secure, but rather sufficiently masked to resist analysis and tampering attacks in a white-box environment, where the attacker has full privileges.

5.3.2 A Network of Crypto Guards

With *crypto guards* as building blocks we can construct a network of code dependencies that make it harder to analyze or modify code statically or dynamically.

A first requirement is protection against static analysis. To this end, all functions in the binary image, except for the entry function, are encrypted with unique dynamic keys. We opted to decrypt the functions just before they are called and to re-encrypt them after they have returned. In this case, only functions on the call stack will be in the clear.

Secondly, as the key for decryption should be fixed, regardless of how the function was reached, we need to know in advance the state of the code from which the key is derived (encrypted or in the clear). Many functions have multiple potential callers. Therefore, we cannot always use the code of the caller to derive the key as the calling functions will be decrypted while the other potential callers remain encrypted. The solution is to use a dominator in the call graph. As a dominator is by definition on the call stack when the function is called, it is guaranteed to be in the clear. A definition is given below.

Definition 3 *Given a function f, a* ***dominator*** *of f is a function which is always present in the paths leading to f in a call graph G, i.e. which is always on the call stack when f gets called.*

We have chosen to use the immediate dominator to derive the key. If a function has exactly 1 caller, this caller represents the immediate dominator.

Definition 4 *Given a set of dominators of f, the* ***immediate dominator*** *is the dominator with the minimal distance to f in a call graph G.*

Note that it is also possible to derive the key of other functions, allowing one to create schemes with different properties. First, one might increase tamper-resistance by checking more code than that from the immediate dominator alone, which offers better code coverage. Secondly, one could construct a scheme to delay failure upon malicious tampering [127]. On the one hand, this may allow a tampered application to run longer, on the other hand this does not correlate the moment of failure to the embedded checking or reaction mechanism.

Thus, a good mode of operation for the encryption (i.e. with error propagation) in combination with our dependency scheme, will propagate errors (triggered by tampering):

- through the modified function due to the mode of operation of the cipher; and
- inheritable from caller to callee due to our dependency scheme (i.e. the key to decrypt the callee depends on the caller's code).

The latter, however, does not validate for multiple callers due to our relaxation (using the dominator's code instead of caller's to derive the key) or to functions with no callees. In theory this could be solved by using authenticated encryption modes, such as EAX [14] or the more efficient OCB [106]. These modes aim to efficiently offer confidentiality and data authentication.

The operation of a function call is illustrated in Figure 5.1. It consists of the following steps:

1. the caller calls a guard to decrypt the callee;
   (a) the guard computes a checksum of the immediate dominator of the callee;
   (b) the callee is decrypted with the checksum as key;
   (c) the guard returns;
2. the caller calls the callee;
3. the callee returns;
4. the caller calls the guard to encrypt the callee;
   (a) the guard computes a checksum of the immediate dominator of the callee;
   (b) the callee is encrypted with the checksum as key;

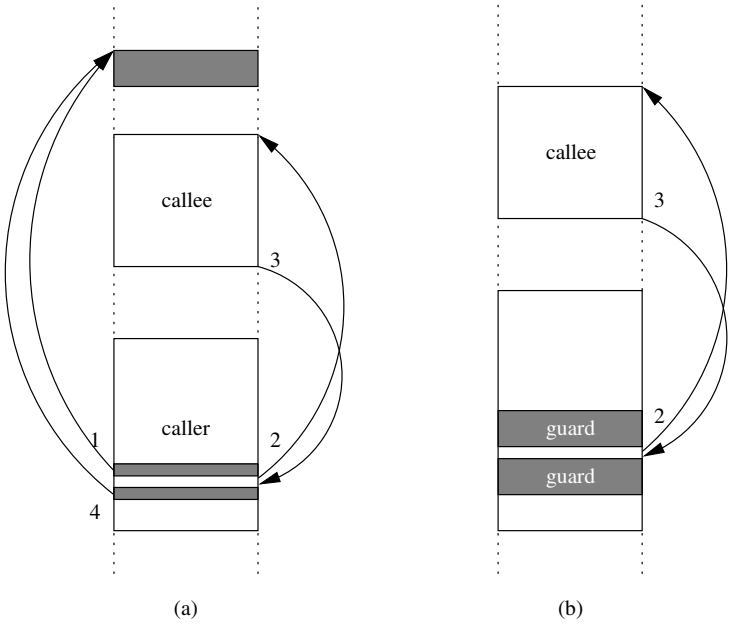


Figure 5.1: (a) Memory layout of function call with calls to a crypto guard prior to the actual call and after its corresponding return. (b) Memory layout after inlining the guard code. Note that the caller's code size will increase relative to the code size of the crypto guard.

(c) the guard returns.

In Cappaert *et al.* [23], we propose two other schemes involving respectively 1 guard, or 4 guards per function call. The former requires an additional check in case a function is repeatedly called to avoid decrypting code which appears to be in the clear already. The latter encrypts the caller from the moment control is transferred to the callee, and decrypts the caller prior to returning of the callee. This results in a memory layout where only the running function (including inlined crypto guards) is susceptible to analysis. Unfortunately, this scheme not only doubles the number of guards, in combination with inlining the crypto guards and depending on the application, performance results might decrease substantially. For more details, we refer to [23]. The rest of this chapter assumes a scheme with 2 guards per call, i.e. a callee is decrypted prior to its call, and re-encrypted after it returns.

5.4 Numerical Evaluation

In our experiments we tested 5 benchmark programs out of the SPEC CPU2006 test suite [122] on an AMD Sempron 1200 MHz, running GNU/Linux with 1 GB of RAM. We first measure the impact of bulk encryption. Subsequently, we apply our on-demand encryption scheme where we protect a maximal number of functions, such that our scheme can offer tamper-resistance according to the properties mentioned in Section 5.3. To insert the guard code we used Diablo [51], a link-time binary rewriter, allowing us to patch binary code, insert extra encryption functionality, and perform dominator analysis on either the control flow graph or the function call graph. As we are generating self-modifying code, we mark all code segments to be readable and writable. Our current implementation only handles functions which respect the `call-return` conventions.

To report the performance cost we define the *time cost* C_t for a program P and its protected version $\bar{P}$ as follows:

$$C_t(P, \bar{P}) = \frac{T(\bar{P})}{T(P)}$$

where $T(X)$ is the execution time of program X.

5.4.1 Bulk Decryption

For the bulk decryption we added a decryption routine that is executed prior to transferring control to the entry point of the program. For simplicity, we encrypted the entire code section of the binary (including library functions as Diablo works on statically compiled binaries). The resulting overhead in execution time is less than 1%.

5.4.2 On-Demand Decryption

On-demand decryption protects functions by decrypting them just before they are called and re-encrypting them after they have returned. Despite the simplicity of this scheme, a number of issues need to be addressed. An overview is given below.

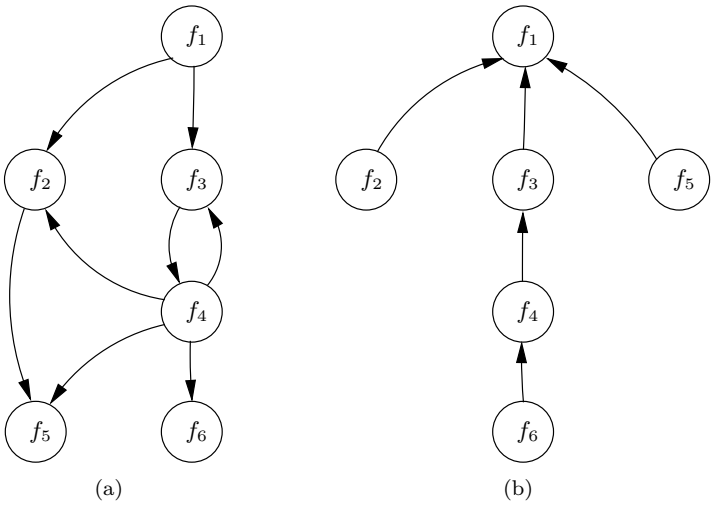


Figure 5.2: (a) A function call graph containing recursive cycle $\{f_3, f_4\}$. Function f_1 gives control to the recursive cycle after decrypting f_3 and f_4. Functions f_5 and f_6 are only decrypted upon calling, however, they are never hashed as they are not calling any other functions. (b) A *dominator tree* where each function has a single directed edge towards its unique *immediate dominator*.

Loops. A scheme considering only decryption should not be nested in a loop (unless it tests for the state – cleartext or ciphertext – of code). Bulk decryption should happen only once. However, the sooner this decryption is performed, the longer code will be exposed. As our scheme operates on a function level and a corresponding function call graph, we do not posses information on loops, unless derived from further analysis.

Recursion. If a function calls itself (pure recursive call), it should – according to our scheme – decrypt itself, although it might be in clear already. Therefore, we suggest decrypting a recursive function only once: namely when it gets called by another function. We can extend this to recursive cycles, where a group of functions together form a recursion. In this case, all functions in the recursive cycle should be decrypted before entering the recursive cycle. Figure 5.2(a) illustrates this. Before giving control to f_3, cycle $\{f_3, f_4\}$ has to be decrypted. Function f_6 can be decrypted (before calling) in f_4 as it will always be re-encrypted (after returning) before f_4 calls it again (e.g. in another iteration of the recursion).

Table 5.1: Time cost for on-demand encryption, using our tamper resistance scheme. This table shows the total number of functions, functions protected with the on-demand encryption scheme, the time cost, and the number of guard pairs (decryption and re-encryption) in the binary.

Program name	Functions total	Functions on-demand	Speed cost C_t	Number of guard pairs
mcf	22	20	1.09	28
milc	159	146	8.17	543
hmmer	234	184	3.20	873
lbm	19	12	1.00	20
sphinx_livepretend	210	192	6.65	1277

Multiple callers. In order to propagate errors through the whole call graph according to the call stack, decryption of a callee should depend on the integrity of all callers. This is not straightforward as we defined our scheme to rely on cleartext code, but at runtime exactly one of the potential callers is in clear. According to Cappaert *et al.* [23], it is possible to decrypt the code of each caller. However, this requires $O(nd)$ decryptions (via a guard) where n represents the number of callers and d the distance in the call graph from the callers to their immediate dominator. To overcome this overhead, we propose to apply a similar strategy as proposed for recursion, namely to decrypt ahead of the call to the caller. Thus, to decrypt a function f_b with multiple callers f_{a_i}, we can decrypt the code of f_b before executing (and decrypting) one of its callers f_{a_i}, and use the (encrypted) code of each caller as a decryption key, yielding a multi-layer decryption: $f_b = D_{h(\bar{f_{a_i}})}(\bar{f_b})$ for all n callers). This would only result in $O(n)$ guards but still propagate errors from caller to callee in the case of multiple callers. Note that decrypting ahead of the actual call to f_b only exposes f_b's code a little longer.

In our implementation we rely on the immediate dominator instead of the actual callers, but we only decrypt the code of f_b when it is called from one of its callers f_{a_i}. The reason is that the dominator is always on the call stack, and thus in the clear, when a function is reached. This only requires $O(n)$ guards.

Table 5.1 shows the time cost after applying our on-demand encryption scheme to 5 benchmark programs out of the SPEC CPU2006 test suite. It is clear that, depending on the nature of the program (number of calls, function size, etc.), the impact of our scheme is moderate for some, while expensive for others.

5.4.3 Combined Scheme

To address the trade-off between performance and protection, we propose a combined scheme. This scheme combines the merits of bulk encryption and the tamper-resistant properties of our on-demand decryption scheme. To decide whether a function is a good candidate for on-demand decryption, we define a heuristic *hotness* that is correlated to the frequency a function is called, namely:

Definition 5 *A function is* ***hot*** *when it is part of the set of most frequently called functions that together contribute to $K\,\%$ of all function calls.*

The call information was collected by analyzing dynamic profile information gathered by Diablo. This definition can be expressed by the following formula:

$$f \text{ is } hot \leftrightarrow calls(f) \geq threshold$$

with

$$threshold = calls(f_i) \mid \sum_{j=1}^{i} calls(f_j) > \frac{K}{100} \sum_{j=1}^{n} calls(f_j)$$

assuming n functions, ordered descending according to the number of times a function f_i is called, i.e. $calls(f_i)$.

When a function is hot, it is not selected for on-demand encryption but protected by bulk encryption. Table 5.2 contains the time costs of the same benchmark programs tested when we apply the combined scheme. It is clear that defining a *hot* threshold reduces the overhead introduced by our guards. We believe that further fine-tuning of our threshold (e.g. increasing the K factor) will improve the performance of all programs.

Furthermore, we also would like to stress that we are aiming to protect all functions at all times, while most other software protection techniques focus on the critical parts only, or all functions but not at all times (e.g. bulk encryption).

Table 5.2: Time cost for our combined scheme, combining on-demand encryption with bulk encryption for $K = 90$.

Program name	Functions total	Functions on-demand	Speed cost C_t	Number of guard pairs
mcf	22	19	1.04	24
milc	159	135	1.95	486
hmmer	234	183	1.15	862
lbm	19	8	1.00	17
sphinx_livepretend	210	181	1.72	1257

5.5 Attacks and Improvements

5.5.1 White-Box Attacks

Our guards, which modify code depending on other code, offer several advantages over the software guards proposed by Chang and Atallah [26] and the those from Horne *et al.* [64]. An overview is given below:

Confidentiality. First, all functions are encrypted statically, either by bulk or by on-demand encryption. An attacker analyzing code statically is forced to first derive all dynamic decryption keys and then decrypt the code. Secondly, as long as code remains encrypted in memory it is protected against memory dump attacks. With a good scheme it is feasible to ensure only a minimal number of code blocks are present in memory in decrypted form. Hence, an attacker dumping memory would only be able to inspect functions part of the call stack. Trading off security for performance, using the hotness heuristic, selects more functions for bulk encryption, hence making them susceptible to dynamic analysis.

Tamper resistance. Together with a good dependency scheme, our guards offer protection against attempts to modify the program code. If a function is tampered with statically or even dynamically, the program will generate corrupted code at a later stage and thus it will eventually crash due to illegal instructions or output unreliable results. Furthermore, if the modification generates executable code, errors will still appear in other functions. An attacker using a debugger to step-trace through the program, might fail as well. For example, the Unix debugger `gdb` [123] uses software breakpoints. These

software breakpoints alter the loaded code in memory. If the corresponding code is either hashed (to derive a decryption key) or decrypted, this will induce faults.

Resistance to a hardware-assisted circumvention attack. An attack, proposed by van Oorschot *et al.* [133], exploits differences between data reads and instruction fetches to bypass self-checksumming code. The attack consists of duplicating each memory page, one page containing the original code, while another contains tampered code. A modified kernel intercepts every data read and redirects it to the page containing the original code, while the code fetched for execution is the modified one. However, more recent work of Giffin *et al.* [59] illustrates that self-modifying code can detect such an attack and thus protect against it. As our work focuses on self-encrypting code, a type of self-modifying code, the detection mechanism of Giffin *et al.* also applies to our technique.

Nevertheless, note that an attacker debugging the program can observe dynamically computed keys at some moment in time. The same counts for addresses of the decryption areas (i.e. function boundaries in our case). Therefore, we propose to protect guards in a diversified manner by obfuscation techniques [141] such that not all of them can be broken in an automated way.

5.5.2 Inlining Guard Code

Embedding a single decryption routine in a binary is not a secure stand-alone protection technique. It should always be combined with other techniques such as obfuscation or self-checking code. The strength of our scheme is a direct consequence of its distributed nature, i.e. a network of guards (as explained in Section 5.3.2). If implementation of the dependency scheme consists of a single instance of the guard code and numerous calls to it, an attacker can modify the guard or crypto code to write all decrypted content to a particular file or memory region. To avoid that an attacker only needs to attack this single instance of the guard code, inlining the entire guard could preclude this attack and force an attacker to modify all instances of the guard code at run time, as all nested guard code will initially be encrypted. This has been illustrated in Figure 5.1(b). However, a disadvantage of this inlining is code expansion. Compact encryption routines might keep the spatial cost relatively low, but implementations of secure cryptographic functions are not always small.

Even though our initial results illustrated in Table 5.1 and Table 5.2 were performed by inlining calls, we expect similar performance results as our guard

code in its most compact form does not exceed 40 bytes, while the calls we used for testing are 47 bytes long (pushing and popping arguments included).

5.5.3 Increasing Granularity and Scheme Extensions

Our scheme is built on top of static call graph information and therefore uses functions as building blocks. If one increases the granularity, and encrypts parts of functions, the guards can be integrated into the program's control flow which will further complicate analyzing the network of guards especially when inlined. However, we believe that such a fine-grained structure will induce much more overhead. The code blocks to be encrypted will be much smaller than the added code. Furthermore, more guards will be required to cover the whole program code. Hence, it is important to trade off the use of these guards, focusing instead on critical parts of the program, and avoiding "hot spots" such as frequently executed code.

As implied by Figure 5.1, the caller remains in cleartext as long as it is part of the call stack. Another extension involves encrypting the caller of a callee when the callee executes. This corresponds to protecting functions on the call stack. As such, only the executing function will be in cleartext. This extension would double the number of guards per original function call inducing considerable overhead, see also Cappaert *et al.* [23]. However, using dedicated heuristics, such as *hotness*, would help us make a better trade-off between on-demand encryption and bulk encryption.

5.6 Conclusions

This chapter presents a technique to protect against dynamic attacks by means of code encryption. Bulk decryption is vulnerable to memory dump attacks once code appears decrypted in memory during execution. To limit the amount of code exposed in memory, we propose an on-demand decryption scheme which operates at function granularity. To improve tamper resistance, we compute decryption keys based on the hash of other code at runtime. This technique is stronger and harder to tamper with than hard-coded keys. We define the dual task of hashing code to derive a key and encrypting/decrypting code with that key as a new type of software guards which offer confidentiality of code, a property that previously proposed software guards did not posses.

Even though encryption is one of the best understood information hiding techniques and has been used previously for software protection, little details

on its practical application and performance impact in the context of software protection are available. To measure performance results, we apply our scheme on 5 programs of the SPEC CPU2006 test suite using Diablo, a link-time binary rewriter. As a security-performance trade-off, we introduce a heuristic based on the frequency that a particular function is called.

The research presented in this chapter was a cooperation with researchers from UGent/ELIS/PARIS and was presented a the ISPEC'08 conference [25]. The initial work defining crypto guards was presented at the WISSec 2006 workshop by Cappaert *et al.* [23], while the algorithm is also included in the book "Surreptitious Software" by Collberg and Nagra [35].

Chapter 6

Conclusions and Future Research

6.1 General Conclusions

More and more software gets distributed over the Internet. Once downloaded, a software owner loses all control over his product. Hosts can no longer be fully trusted as users are in full control of their machine, both software and hardware. Malicious users can attack software in multiple ways. First, they can inspect software by performing *reverse-engineering* techniques. Static analysis techniques include disassemblation and decompilation. Dynamic analysis techniques observe program behavior at runtime, similar to debugging. Secondly, users with malicious intentions might alter software to meet their own needs. Often these *tampering* attacks are performed to circumvent built-in security code that prevents violation of license agreements, etc.

Code obfuscation represents a whole set of software protection techniques that raise the bar for the attacker in terms of skills or resources. They are inexpensive to deploy compared to hardware-assisted solutions, as no additional hardware needs to be acquired. They are flexible as techniques can be applied to a certain degree that meets security versus performance trade-offs set by the software owner. Furthermore, obfuscation techniques can be easily integrated during software development, as the techniques are software-only.

In this thesis, we examined software protection techniques to obfuscate code. In Chapter 3, we compare several state-of-the-art protection techniques based

on their potential to protect against static/dynamic analysis or tampering. Subsequentially, we propose two new techniques which we summarize below.

Protection against static control flow analysis. In Chapter 4, we present a model to strengthen flattened control flow graphs against static analysis. Wang *et al.* [138, 137] introduce the concept of *control flow graph flattening* and present aliasing as a mean to make static analysis harder. Our contributions are as follows:

- Our general model makes reconstruction of the original control flow graph harder. It forces the attacker to analyze a larger code portion of code in order to understand the program control flow. In Section 4.4.2, we gradually strengthen the results from László and Kiss [80]. For this we use one-way functions and functions which do not leak control flow information.
- Our model shifts the secrecy of program control flow to a smaller component: a secret key or secret function. This allows our model to be easily mapped onto hardware-assisted solutions. Furthermore, depending on the existence of *opaque predicates* [42], we propose a software-only solution as well. In combination with strong opaque predicates, our scheme is resistant to static analysis attacks that try to extract a program's control flow graph.

To examine the strength of our model, we present several attacks and argument how our scheme resists those static analysis attacks.

Protection against dynamic analysis and tampering attacks. In Chapter 5, we define a new type of software guard, named *crypto guard*, which protects *confidentiality* and *integrity* of program code. Our contributions are briefly summarized below.

- Our software guard protects against program analysis (i.e. protecting the confidentiality of code). This is achieved by on-the-fly decryption and encryption of functions to minimize exposure to static and dynamic program inspection. A function is only decrypted prior to its call. We denote the latter as *on-demand* decryption.
- Tamper resistance or integrity of program code is achieved by (1) creating a web of dependencies, and (2) using non-malleable cryptography. The first is achieved by using encryption/decryption keys based on hashes

of other code fragments. The latter is achieved by the use of strong cryptography such that changing encrypted code does not result in usable plaintext code.

Our crypto guards hereby extend the work from Chang and Atallah [26] and from Horne *et al.* [64].

To illustrate the usability of our protection scheme, we apply our technique to several benchmark programs of the SPEC CPU2006 test suite cpu2006. In order to further trade off security versus performance, we propose a *hotness* heuristic. This allows the developer to select a function for on-demand decryption or bulk decryption which reduces runtime overhead.

6.2 Future Work

In this section, we propose several directions for further work. However, as software protection and defense is a highly interactive area, focuses might shift and new research challenges might appear.

Specialized heuristics. In Chapter 4, we present a scheme built on top of control flow graph flattening. Laszló and Kiss [80] illustrate that flattening itself on average doubles execution time of C++ programs. Implementing our scheme on top of control flow graph flattening would introduce extra overhead based on the branch functions $B_i()$ and the transition function $F()$. Namely, each control transfer a branch function $B_i()$ is computed followed by a pass through the transition function $F()$. In case of small basic blocks BB_i but a relatively large function $F()$ we expect considerable overhead in execution time. Therefore, it might be appropriate to either apply our technique to a critical component in an application, or to introduce heuristics that group basic blocks to larger code blocks which then can be protected using our scheme.

In Chapter 5, we define a *hotness* heuristic which decides whether a function body will be decrypted on-demand (i.e. just before the call in our case), or by bulk decryption (i.e. at the time the program is loaded in memory). Hence, any memory dump after the program is loaded in memory, would reveal the code of these *hot* functions. However, there are several alternatives to improve this. Consider a call to a function f in a frequently executed loop. If our heuristic classifies f as *hot*, it will be decrypted at the time the program is loaded. But it might be sufficient to just decrypt it outside the loop. Nevertheless, this requires extra information on the semantics of the program to be combined with accurate profile information. A good level to gather information on the

semantics of a program would be the compiler level where profile information could be fed back into a compiler (e.g. LLVM [2]). Extra heuristics would definitely help here to trade off security versus performance.

Security metrics. In the code obfuscation field, suitable metrics are still an open research domain. Every new proposal in the software security scene is welcomed with a healthy dose of scepticism. In contrast to the field of cryptography, code obfuscation has no rich background in measuring strength and proving properties. The work from Collberg *et al.* [37] presents some metrics on potency, resilience, and stealth to express the strength of a transformation. However, these software metrics are still hard to map onto complexity results, expressing how hard it is for a human attacker or automated program to perform a specific attack. Nevertheless, in the field of cryptography, the time or memory complexity of an attack in the field of cryptography is expressed as a power of 2. Depending on the evolutions in hardware and software, this attack can then be considered feasible or not within a certain time span.

In order to achieve similar results in the field of software security, we propose structured solutions rather than "spaghetti code" solutions, where applications are applied at random. While the latter might obscure data for human inspection, it usually does not add much complexity for automated tools. In the case of structured solutions, security properties might be easier to prove and measure. Furthermore, structured code is easier in terms of maintenance than spaghetti code.

An example of a structured code transformation is Wang's control flow graph flattening [138, 137]. The complexity of a control flow graph is related to the number of possible paths through the graph, and in a flattened graph this number of (statically) possible paths is maximized. McCabe's cyclomatic number [92] is a metric that corresponds to the number of decisions points plus one. In a flattened control flow graph this number is also maximized. Laszló and Kiss [80] observed that the cyclomatic number increases by a factor 2 to 5 when flattening C++ benchmark programs. Our scheme with uniform branch functions $B_i()$ and a fixed transition function $F()$ is built on top of these flattened graphs. However, we only protect against control flow analysis. A structured way to obfuscated data flow could be to make data flow analysis highly flow-sensitive by using pointers that are manipulated in a way related to the technique we use in Section 4.4.2 to update the switch variable.

Hardware support. Software-based techniques can always benefit from hardware support. Therefore, software protection techniques should be designed in a modular fashion, so that integrating hardware support can act as a

complement to the strength of the protection technique. In Chapter 5, we heavily rely on code encryption and cryptographic hash functions. Hardware support such as Intel's AES instruction set [62], can ensure a performance boost to future solutions relying on the Advanced Encryption Standard [46].

Since 2004, many major PC manufacturers have shipped systems that have a trusted platform module (TPM) inside. If enabled, this TPM allows a third party to verify the integrity of hardware and software running on the device. The inclusion of this trusted hardware module was one of the features requested by media companies to facilitate DRM technology in order to enforce their rules. However, this approach is considered controversial by the software community as users would no longer be able to freely modify all software running on the machine. Nevertheless, we believe that small, trusted hardware components could be used as cornerstones to build protection techniques. For example, Schellekens *et al.* [108] present a remote attestation technique using a TPM's trusted clock.

Combinations of software protection techniques. In Chapter 3, we gave an overview of software protection techniques. Despite not all techniques being combinable with others, it is clear that a combination of techniques offers stronger solutions. In '96, Aucsmith [11] proposed his model for "tamper resistance software" using encryption, digital signatures, etc.

Many techniques require 'stealth' code, which blends in with other program code. Security of software guards [26, 64] and crypto guards (see Chapter 5) assumes that guards cannot be easily identified and tampered with themselves. Hence, code obfuscation transformations could diversify software guards or make them more analysis/tamper resistant.

Although not applicable to ciphers only, white-box cryptography [144] is a technique that makes it hard to extract a key embedded in a cryptographic cipher implementation. As the cipher is replaced by an abstract network of lookup tables and the key is spread over many lookup tables, key extraction becomes rather a dynamic algebraic attack than a static code analysis attack. Replacing code portions or operations by lookup tables, could be seen as a way to obfuscate code. Michiels *et al.* [95] use white-box cryptography to create tamper resistant code. The ability to inject randomness in lookup tables allows them to use lookup tables both as data and as code. Tampering with one would break the other. In general, we believe that future software protection research could benefit from white-box cryptography ideas or other related concepts.

Application domains. In Chapter 2, we presented some applications domains. These include applications enforcing digital rights management, distributed network applications, and mobile software agents. With the current evolution in hardware and software, users shift from desktop and servers to mobile devices, such as smartphones and tablet computers. Simultaneously, attackers invent methods to exploit weaknesses in these mobile devices and their apps. Future software security research should focus on light-weight solutions, suitable to run on embedded and mobile devices with limited resources. Davi *et al.* [50] present a method based on control-flow integrity to protect against runtime attacks on smartphones. They implemented their solution on Apple's iOS platform to illustrate its low impact on performance.

Bibliography

[1] Java 2 platform security architecture. http://docs.oracle.com/javase/1.4.2/docs/guide/security/spec/security-spec.doc.html (consulted on February 10th, 2012).

[2] The LLVM compiler infrastructure. http://llvm.org (consulted on February 10th, 2012).

[3] The International Obfuscated C Code Contest. http://www.ioccc.org/ (consulted on February 10th, 2012).

[4] Trusted computing group. http://www.trustedcomputinggroup.org/ (consulted on February 10th, 2012).

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] B. Anckaert. *Diversity for Software Protection*. PhD thesis, Ghent University, 2008.

[7] B. Anckaert, B. De Sutter, D. Chanet, and K. De Bosschere. Steganography for executables and code transformation signatures. In C. Park and S. Chee, editors, *ICISC*, volume 3506 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2004.

[8] B. Anckaert, M. H. Jakubowski, and R. Venkatesan. Proteus: virtualization for diversified tamper-resistance. In M. Yung, K. Kurosawa, and R. Safavi-Naini, editors, *Digital Rights Management Workshop*, pages 47–58. ACM, 2006.

[9] B. Anckaert, M. H. Jakubowski, R. Venkatesan, and C. W. Saw. Runtime protection via dataflow flattening. In R. Falk, W. Goudalo, E. Y. Chen, R. Savola, and M. Popescu, editors, *SECURWARE*, pages 242–248. IEEE Computer Society, 2009.

[10] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In G. Karjoth and K. Stølen, editors, *QoP*, pages 15–20. ACM, 2007.

[11] D. Aucsmith. Tamper resistant software: An implementation. In R. J. Anderson, editor, *Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer, 1996.

[12] F. B. and Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993. http://all.net/books/IP/evolve.html (consulted on February 10th, 2012).

[13] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Advances in Cryptology - Crypto 2001*, LNCS 2139:1–18, 2001.

[14] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. In B. K. Roy and W. Meier, editors, *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407. Springer, 2004.

[15] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[16] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In B. S. K. Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

[17] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white box AES implementation. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 227–240. Springer, 2004.

[18] P. Biondi and F. Desclaux. Silver needle in the Skype. *Black Hat Europe'06*, March 2006.

[19] W. Blume and R. Eigenmann. Symbolic range propagation. *In Proceedings of the 9th international Symposium on Parallel Processing (IPPS), IEEE Computer Society*, pages 357–363, April 1995.

[20] F. Boldewin. The big SoftICE howto. http://www.reconstructer.org/papers/The%20big%20SoftICE%20howto.pdf (consulted on February 10, 2012).

[21] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In W. Fumy, editor, *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[22] J. Bringer, H. Chabanne, and E. Dottax. Perturbing and protecting a traceable block cipher. In H. Leitold and E. P. Markatos, editors, *Communications and Multimedia Security*, volume 4237 of *Lecture Notes in Computer Science*, pages 109–119. Springer, 2006.

[23] J. Cappaert, N. Kisserli, D. Schellekens, and B. Preneel. Self-encrypting code to protect against analysis and tampering. *1st Benelux Workshop on Information and System Security (WISSec 2006)*, page 14, 2006.

[24] J. Cappaert and B. Preneel. A general model for hiding control flow. In E. Al-Shaer, H. Jin, and M. Joye, editors, *Digital Rights Management Workshop*, pages 35–42. ACM, 2010.

[25] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. De Bosschere. Towards tamper resistant code encryption: Practice and experience. In L. Chen, Y. Mu, and W. Susilo, editors, *ISPEC*, volume 4991 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2008.

[26] H. Chang and M. J. Atallah. Protecting software code by guards. In T. Sander, editor, *Digital Rights Management Workshop*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2001.

[27] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 559–572. ACM, 2010.

[28] H.-Y. Chen, T.-W. Hou, and C.-L. Lin. Tamper-proofing basis path by using oblivious hashing on Java. *SIGPLAN Not.*, 42(2):9–16, February 2007.

[29] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In F. A. P. Petitcolas, editor, *Information Hiding*, volume 2578 of *Lecture Notes in Computer Science*, pages 400–414. Springer, 2002.

[30] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. White-box cryptography and an AES implementation. In K. Nyberg and H. M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 250–270. Springer, 2002.

[31] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. A white-box DES implementation for DRM applications. In J. Feigenbaum, editor, *Digital Rights Management Workshop*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.

[32] S. Chow, Y. X. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In G. I. Davida and Y. Frankel, editors, *ISC*, volume 2200 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2001.

[33] C. Cifuentes and K. Gough. Decompiling of binary programs. *Software – Practice & Experience*, 25(7):811–829, 1995.

[34] J. Claessens, B. Preneel, and J. Vandewalle. (How) Can Mobile Agents Do Secure Transactions on Untrusted Hosts? – A Survey of the Security Issues and the Current Solutions. *ACM Transactions on Internet Technology*, 3(1):28–48, February 2003.

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

[36] C. Collberg, J. Nagra, and F.-Y. Wang. Surreptitious software: Models from biology and history. In V. Gorodetsky, I. Kotenko, and V. A. Skormin, editors, *Computer Network Security*, volume 1 of *Communications in Computer and Information Science*, pages 1–21. Springer Berlin Heidelberg, 2007.

[37] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report #148, Department of Computer Science, The University of Auckland, 1997. http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/A4.pdf (consulted on February 10th, 2012).

[38] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *International Conference on Computer Languages*, pages 28–38, 1998.

[39] C. S. Collberg, E. Carter, S. K. Debray, A. Huntwork, J. D. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In W. Pugh and C. Chambers, editors, *PLDI*, pages 107–118. ACM, 2004.

[40] C. S. Collberg, A. Huntwork, E. Carter, G. M. Townsend, and M. Stepp. More on graph theoretic software watermarks: Implementation, analysis, and attacks. *Information & Software Technology*, 51(1):56–67, 2009.

[41] C. S. Collberg and C. D. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. volume 28, pages 735–746, 2002.

[42] C. S. Collberg, C. D. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL*, pages 184–196, 1998.

[43] K. Coogan, G. Lu, and S. K. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 275–284. ACM, 2011.

[44] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade, 2000.

[45] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[46] J. Daemen and V. Rijmen. The design of Rijndael: AES - the Advanced Encryption Standard, 2002.

[47] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque predicates detection by abstract interpretation. In *Proc. of the 11th International Conference on Algebraic Methodology and Software Technology* (*AMAST '06*), volume 4019 of *Lecture Notes in Computer Science*, pages 81–95. Springer-Verlag, 2006.

[48] L. D'Anna, B. Matt, A. Reisse, T. V. Vleck, S. Schwab, and P. Leblanc. Self-protecting mobile agents obfuscation report – final report. Technical Report #03-015, Network Associates Laboratories, July 2003.

[49] Datarescue. IDA Pro. http://www.datarescue.com/idabase/ (consulted on February 10th, 2012).

[50] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. Poster: control-flow integrity for smartphones. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 749–752. ACM, 2011.

[51] B. De Sutter, L. Van Put, D. Chanet, B. De Bus, and K. De Bosschere. Link-time compaction and optimization of ARM executables. *ACM Trans. Embedded Comput. Syst.*, 6(1), 2007.

[52] S. K. Debray and J. Patel. Reverse engineering self-modifying code: Unpacker extraction. In G. Antoniol, M. Pinzger, and E. J. Chikofsky, editors, *WCRE*, pages 131–140. IEEE Computer Society, 2010.

[53] 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*, IH'07, pages 80–95, Berlin, Heidelberg, 2007. Springer-Verlag.

[54] Y. Dodis, A. Elbaz, R. Oliveira, and R. Raz. Improved randomness extraction from two independent sources. *International Workshop on Randomization and Approximation, Techniques in Computer Science (RANDOM)*, August 2004.

[55] R. El-Khalil and A. D. Keromytis. Hydan: Hiding information in program binaries. In J. Lopez, S. Qing, and E. Okamoto, editors, *ICICS*, volume 3269 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2004.

[56] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[57] J. A. Garay and L. Huelsbergen. Software integrity protection using timed executable agents. In F.-C. Lin, D.-T. Lee, B.-S. Lin, S. Shieh, and S. Jajodia, editors, *2006 ACM Symposium on Information, Computer and Communications Security (ASIACCS 2006), Taipei, Taiwan, March 21-24, 2006*, pages 189–200. ACM, 2006.

[58] S. Ghosh, J. D. Hiser, and J. W. Davidson. A secure and robust approach to software tamper resistance. In *Proceedings of the 12th international conference on Information hiding*, IH'10, pages 33–47, Berlin, Heidelberg, 2010. Springer-Verlag.

[59] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSA05)*, pages 23–32. IEEE Computer Society, 2005.

[60] L. Goubin, J.-M. Masereel, and M. Quisquater. Cryptanalysis of white box DES implementations. In *Proceedings of the 14th International Workshop on Selected Areas in Cryptography (SAC 2007)*, volume 4876 of *Lecture Notes in Computer Science*, pages 278–295. Springer-Verlag, 2007.

[61] Y. X. Gu, B. Wyseur, and B. Preneel. Software-based protection is moving to the mainstream. *IEEE Software, Special Issue on Software Protection*, 28(2):56–59, 2011.

[62] S. Gueron. Intel advanced encryption standard (AES) instructions set, January 2010.
http://software.intel.com/file/24917 (consulted on February 10th, 2012).

[63] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. volume 52, pages 91–98, 2009.

[64] B. G. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In T. Sander, editor, *Digital Rights Management Workshop*, volume 2320 of *Lecture Notes in Computer Science*, pages 141–159. Springer, 2001.

[65] M. Howard and D. C. LeBlanc. *Writing Secure Code, Second Edition.* Microsoft Press, 2002.

[66] M. Jacob, D. Boneh, and E. W. Felten. Attacking an obfuscated cipher by injecting faults. In J. Feigenbaum, editor, *Digital Rights Management Workshop*, volume 2696 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2002.

[67] 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*, MM&Sec '07, pages 129–140, New York, NY, USA, 2007. ACM.

[68] M. Jakobsson and M. K. Reiter. Discouraging software piracy using software aging. In T. Sander, editor, *Digital Rights Management Workshop*, volume 2320 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2001.

[69] M. H. Jakubowski, P. Naldurg, V. Patankar, and R. Venkatesan. Software integrity checking expressions (ices) for robust tamper detection. In T. Furon, F. Cayre, G. J. Doërr, and P. Bas, editors, *Information Hiding*, volume 4567 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2007.

[70] M. H. Jakubowski, C. W. Saw, and R. Venkatesan. Iterated transformations and quantitative metrics for software protection. In E. Fernández-Medina, M. Malek, and J. Hernando, editors, *SECRYPT*, pages 359–368. INSTICC Press, 2009.

[71] M. H. Jakubowski, C. W. Saw, and R. Venkatesan. Tamper-tolerant software: Modeling and implementation. In T. Takagi and M. Mambo,

editors, *IWSEC*, volume 5824 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2009.

[72] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *COMPSAC*, pages 170–179. IEEE Computer Society, 2003.

[73] S. T. Kelly. Analysis of Code Red, 2001. http://www.thehackademy.net/madchat/vxdevl/papers/avers/paper32.pdf (consulted on February 10th, 2012).

[74] A. Kerckhoffs. La cryptography militaire. *Journal des sciences militaires*, IX:5–38, Janvier 1883. http://www.petitcolas.net/fabien/kerckhoffs/ (consulted on February 10th, 2012).

[75] Z. KlassMaster. The second generation Java obfuscator. http://www.zelix.com/ (consulted on February 10th, 2012).

[76] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[77] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[78] C. Krügel, W. K. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, pages 255–270. USENIX, 2004.

[79] W. Landi. Undecidability of static analysis. *LOPLAS*, 1(4):323–337, 1992.

[80] T. László and A. Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando, Eötvös Nominatae, Sectio Computatorica*, 30:3–19, Agust 2009.

[81] C. Liem, Y. X. Gu, and H. Johnson. A compiler-based infrastructure for software-protection. In Ú. Erlingsson and M. Pistoia, editors, *PLAS*, pages 33–44. ACM, 2008.

[82] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[83] H. E. Link and W. D. Neumann. Clarifying obfuscation: Improving the security of white-box encoding. *Cryptology ePrint Archive*. http://eprint.iacr.org/2004/025.pdf (consulted on February 10th, 2012).

[84] C. Linn and S. K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM Conference on Computer and Communications Security*, pages 290–299. ACM, 2003.

[85] D. Low. Java control flow obfuscation. Master's thesis, University of Auckland, New Zealand, 1998.

[86] 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 H. R. Arabnia and H. Reza, editors, *Software Engineering Research and Practice*, pages 527–533. CSREA Press, 2006.

[87] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In R. Safavi-Naini and M. Yung, editors, *Digital Rights Management Workshop*, pages 75–82. ACM, 2005.

[88] M. Madou, B. Anckaert, P. Moseley, S. K. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In J. Song, T. Kwon, and M. Yung, editors, *WISA*, volume 3786 of *Lecture Notes in Computer Science*, pages 194–206. Springer, 2005.

[89] R. E. Mahan. Malicious software. http://www.tricity.wsu.edu/htmls/cs427/public_html/Ch%2013%20Malicious%20Software.pdf (consulted on February 10th, 2012).

[90] M. Mambo, T. Murayama, and E. Okamoto. A tentative approach to constructing tamper-resistant software. In *Proceedings of New Security Paradigms Workshop*, pages 23–33, 1997.

[91] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[92] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[93] N. Mehta and S. Clowes. *Shiva – Advances in ELF Binary Encryption.* http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-mehta/bh-us-03-mehta.pdf (consulted on February 10th, 2012).

[94] A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[95] W. Michiels and P. Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In *Proceedings of 7th ACM Workshop on Digital Rights Management (DRM 2007)*, pages 82–89. ACM Press, 2007.

[96] Microsoft Corporation. Driver signing requirements for Windows, 2002. http://msdn.microsoft.com/en-us/windows/hardware/gg487317 (consulted on February 10th, 2012).

[97] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm, 2003. http://www.caida.org/publications/papers/2003/sapphire/sapphire.html (consulted on February 10th, 2012).

[98] Y. D. Mulder, B. Wyseur, and B. Preneel. Cryptanalysis of a perturbated white-box AES implementation. In G. Gong and K. C. Gupta, editors, *INDOCRYPT*, volume 6498 of *Lecture Notes in Computer Science*, pages 292–310. Springer, 2010.

[99] V. Nagarajan, R. Gupta, M. Madou, X. Zhang, and B. De Sutter. Matching control flow of program versions. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007)*. IEEE, 2007.

[100] K. Natvig. Sandbox technology inside AV scanners. *Virus Bulletin Conference*, pages 475–488, September 2001. http://www.thehackademy.net/madchat/vxdevl/library/Sandbox%20Technology%20Inside%20AV%20Scanners.pdf (consulted on February 10th, 2012).

[101] K. Natvig. Sandbox II: Internet. *Virus Bulletin Conference*, pages 1–18, September 2002. http://www.windowsecurity.com/uplarticle/20/Sandbox2_vb2002.pdf (consulted on February 10th, 2012).

[102] M. F. Oberhummer and L. Molnar. UPX - the Ultimate Packer for eXecutables. http://upx.sourceforge.net/ (consulted on February 10th, 2012).

[103] B. Page. A report on the Internet worm, 1988. http://www.ee.ryerson.ca/~elf/hack/iworm.html (consulted on February 10th, 2012).

[104] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 19:1–19:16, Berkeley, CA, USA, 2007. USENIX Association.

[105] T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in Java (Does bytecode reveal source?). In *COOTS*, pages 185–198. USENIX, 1997.

[106] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.

[107] T. Sander and C. F. Tschudin. On software protection via function hiding. *In Proceedings of the Second Workshop on Information Hiding*, LNCS 1525:111–123, 1998.

[108] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 74(1-2):13–22, 2008.

[109] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. In A. van Deursen and E. Burd, editors, *WCRE*, pages 45–54. IEEE Computer Society, 2002.

[110] T. T. Scud. ObjObf - x86/Linux ELF relocateable object obfuscator, 2003. http://packetstormsecurity.org/files/31524/objobf-0.5.0.tar.bz2 (consulted on February 10th, 2012).

[111] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. K. Khosla. Externally verifiable code execution. *Commun. ACM*, 49(9):45–49, 2006.

[112] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In A. Herbert and K. P. Birman, editors, *SOSP*, pages 1–16. ACM, 2005.

[113] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–. IEEE Computer Society, 2004.

[114] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[115] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In V. Atluri, B. Pfitzmann, and P. D. McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 298–307. ACM, 2004.

[116] A. Shamir and N. van Someren. Playing "hide and seek" with stored keys. 1648:118–124, 1999.

[117] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 94–109, Washington, DC, USA, 2009. IEEE Computer Society.

[118] I. Simon. A comparative analysis of methods of defense against buffer overflow attacks, January 2000. http://www.mcs.csuhayward.edu/~simon/security/boflo.html (consulted on February 10th, 2012).

[119] S. W. Smith. Secure coprocessing applications and research issues. In *Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory*, 1996.

[120] P. Solutions. DashO - the premier Java obfuscator and efficiency enhancing tool. http://www.preemptive.com/products/dasho/ (consulted on February 10th, 2012).

[121] P. Solutions. Dotfuscator - the premier .NET obfuscator and efficiency enhancing tool. http://www.preemptive.com/products/dotfuscator/ (consulted on February 10th, 2012).

[122] SPEC – Standard Performance Evaluation Corporation. *SPEC CPU2006*. http://www.spec.org/cpu2006/ (consulted on February 10th, 2012).

[123] R. Stallman, R. Pesch, and S. Shebs. Debugging with gdb: The GNU source-level debugger, 2010.

[124] J. P. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater. Robust object watermarking: Application to code. In A. Pfitzmann, editor, *Information Hiding*, volume 1768 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1999.

[125] Symantic. Understanding and managing polymorphic viruses. http://www.symantec.com/avcenter/reference/striker.pdf (consulted on February 10th, 2012).

[126] P. Ször and P. Ferrie. Hunting for metamorphic, September 2001. http://www.peterszor.com/metamorp.pdf (consulted on February 10th, 2012).

[127] G. Tan, Y. Chen, and M. H. Jakubowski. Delayed and controlled failures in tamper-resistant software. In J. Camenisch, C. S. Collberg, N. F. Johnson, and P. Sallee, editors, *Information Hiding*, volume 4437 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2006.

[128] T. Teso. Burneye ELF encryption program 1.0.1, 2002. http://packetstormsecurity.org/files/30648/burneye-1.0.1-src.tar.bz2 (consulted on February 10th, 2012).

[129] C. D. Thomborson, J. Nagra, R. Somaraju, and C. He. Tamper-proofing software watermarks. In J. M. Hogan, P. Montague, M. K. Purvis, and C. Steketee, editors, *ACSW Frontiers*, volume 32 of *CRPIT*, pages 27–36. Australian Computer Society, 2004.

[130] Tool Interface Standards. Executable and Linkable Format (ELF), 1993.

[131] J. D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[132] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *WCRE*, pages 45–54. IEEE Computer Society, 2005.

[133] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Trans. Dependable Sec. Comput.*, 2(2):82–92, 2005.

[134] J. Viega and M. Messier. *Secure Programming Cookbook for C and C++*. O'Reilly Media, Inc., 2003.

[135] Z. Vrba. cryptexec: Next-generation run-time binary encryption using on-demand function extraction, August 2005.

[136] Z. Vrba, P. Halvorsen, and C. Griwodz. Program obfuscation by strong cryptography, 2010.

[137] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Department of Computer Science, University of Virginia, 2000.

[138] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson. Protection of software-based survivability mechanisms. In *DSN*, pages 193–202. IEEE Computer Society, 2001.

[139] R. N. Williams. A painless guide to CRC error detection algorithms, 1993. http://www.repairfaq.org/filipg/LINK/F_crc_v3.html (consulted on February 10th, 2012).

[140] Wired. iPod copy protection cracked, 2006. http://www.wired.com/science/discoveries/news/2006/10/72004 (consulted on February 10th, 2012).

[141] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[142] 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*, CCS '10, pages 536–546, New York, NY, USA, 2010. ACM.

[143] G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy*, pages 127–138. IEEE Computer Society, 2005.

[144] B. Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, 2009.

[145] 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 Workshop on Selected Areas in Cryptography (SAC 2007)*, volume 4876 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, 2007.

[146] B. Wyseur and B. Preneel. Condensed white-box implementations. In *Proceedings of the 26th Symposium on Information Theory in the Benelux (BSIT 2005)*, pages 296–301, Brussels, Belgium, 2005. Werkgemeenschap voor Informatie- en Communicatietheorie.

[147] T. Yetiser. Polymorphic viruses. http://vx.netlux.org/texts/html/polymorf.html (consulted on February 10th, 2012).

[148] K. Zetter. How digital detectives deciphered Stuxnet, the most menacing malware in history, July 2011. http://www.wired.com/threatlevel/2011/07/how-digital-detectives-deciphered-stuxnet/all/1 (consulted on February 10th, 2012).

List of Publications

International articles

1. J. Cappaert, B. Preneel, "A General Model for Hiding Control Flow," *Proceedings of the 10th ACM workshop on Digital Rights Management (DRM 2010)*, ACM, pp. 35–42, 2010.
2. J. Cappaert, B. Preneel, B. Anckaert, M. Madou, K. De Bosschere, "Towards Tamper Resistant Code Encryption: Practice and Experience," *Information Security Practice and Experience, 4th International Conference, ISPEC 2008, Lecture Notes in Computer Science 4991*, L. Chen, Y. Mu, W. Susilo, Eds., Springer-Verlag, pp. 86–100, 2008.
3. N. Kisserli, J. Cappaert, B. Preneel, "Software Security Through Targeted Diversification," *CoBaSSA 2007 proceedings*, 4 pages, 2007.
4. M. Madou, B. Anckaert, B. De Bus, K. De Bosschere, J. Cappaert, B. Preneel, "On the Effectiveness of Source Code Transformations for Binary Obfuscation," *Proceedings of 2006 International Conference on Software Engineering Research and Practice (SERP 2006)*, F. Arbab, H. Reza, Eds., CSREA Press, pp. 527–533, 2006.

National articles

5. Y. De Mulder, J. Cappaert, N. Kisserli, N. Mavrogiannopoulos, B. Preneel, "Perturbated Functions: a new approach to Obfuscation and Diversity," *5th Benelux Workshop on Information and System Security (WISSec 2010)*, 11 pages, 2010.
6. J. Cappaert, N. Kisserli, D. Schellekens, B. Preneel, "Self-encrypting Code to Protect against Analysis and Tampering," *1st Benelux Workshop on Information and System Security (WISSec 2006)*, 14 pages, 2006.

Internal reports

7. J. Cappaert, B. Wyseur, B. Preneel, "Software Security Techniques," COSIC internal report, 42 pages, 2004.

Project reports

8. B. Wyseur, J. Cappaert, M. Ceccato, S. Di Carlo, "Protection mechanisms for hardening the software application against analysis and tampering," RE-TRUST Deliverable D2.4, 16 pages, 2008.

Others

9. J. Cappaert, B. Preneel, "On the Importance of Code Obfuscation," *Program Acceleration through Application and Architecture driven Code Transformations, 4th Symposium*, pp. 91–93, 2004.

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Electrical Engineering (ESAT)
Computer Security and Industrial Cryptography (COSIC)
Kasteelpark Arenberg 10, bus 2446
B-3001 Heverlee (Belgium)