The Pennsylvania State University
The Graduate School

PROTECTING SOFTWARE FROM ATTACK AND THEFT VIA PROGRAM ANALYSIS

A Dissertation in
Computer Science and Engineering
by
Xinran Wang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2009

The dissertation of Xinran Wang was reviewed and approved* by the following:

Sencun Zhu
Assistant Professor of Computer Science and Engineering
Assistant Professor of Information Sciences and Technology
Dissertation Advisor, Chair of Committee

Peng Liu
Associate Professor of Information Sciences and Technology

Guohong Cao
Professor of Computer Science and Engineering

Zan Huang
Assistant Professor of Supply Chain and Information Systems

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Along with the rapid developing software industry and the advent of the Internet, attack and theft are becoming two serious threats to software and software community. Although some attack or theft detection approaches have been proposed, these approaches are limited to meet several highly desired requirements. For example, both attack and theft detection approaches should be resilient to code obfuscation techniques; attack detection approaches should detect new or unknown attacks; software theft detection approaches should be able to detect software component theft.

In this dissertation, several new program analysis techniques, which meet these key requirements, are proposed to detect attack and theft. First, a novel program analysis technique called *code abstraction* is proposed which is a generic method to separate code from data. Based on this technique, an attack detection system called SigFree is designed and implemented. SigFree is signature free, thus it can block new and unknown attacks. Detection effectiveness and performance are evaluated in experiments and the applicability of SigFree is discussed. Second, a static taint and initialization analyses based approach is presented. Compared with existing static analysis approaches developed for the same purpose, the new approach is the first one that can detect attack code obfuscated by self-modifying and indirect jump, and a more comprehensive static analysis solution in defending against advanced obfuscation including anti-signature, anti-static-analysis and anti-emulation code obfuscation. Finally, a system call dependence graph based software birthmark is proposed to identify software theft. A dynamic analysis tool which generates system call dependence graph at run-time is designed and built. We demonstrate the strength of the birthmarks against various evasion techniques, including those based on different compilers and compiler optimization levels as well as state-of-the-art obfuscation tools. Unlike the existing works that were evaluated through toy software, we evaluate our birthmarks on a set of large software.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to my advisor Dr. Sencun Zhu for his inspiration, patience, encouragement and invaluable guidance that helped me in various fields including my study, my research, my scientific presentation and writing, and my creative thinking. Without his tremendous support and constant guidance, I could not have completed this dissertation.

I would also like to thank Dr. Peng Liu , Dr. Guohong Cao and Dr. Zan Huang for their guidance and serving as my committee members. Specifically, I am deeply grateful to Dr. Peng Liu for the inspiration and insightful comments on my thesis, which ensured me to work on the right track towards finishing my dissertation.

My gratitude also goes to my co-author Yoon-Chan Jhi for inspiring conversations and insightful suggestions to help me to work through our research problems. I would like to thank my co-author Chi-Chun Pan for his hard work that led to the development of SigFree system. I am also grateful to the members of Penn State Cyber Security Lab for collecting real traces.

Last but not least, I am deeply indebted to my parents and brothers for their unconditional love and their understanding. I also wish to express my deepest gratitude to my wife for her constant faith in me and sacrifices that she made during my candidature.

This work was supported in part by the CAREER NSF-0643906.

Dedication

To my wife Yi and daughter Sarah.

Chapter 1

Introduction

Software is permeating in modern society. We rely on software not only for business but also for everyday life. Unfortunately, there are a great number of vulnerabilities in today's software. Attack for the widely existed vulnerabilities can cause huge damages. For example, the infamous worm Code Red attacks 359,000 machines exploiting a vulnerability in Microsoft Internet information service in 14 hours and costs $2.6 billion worldwide [35]. On the other hand, along with the rapid developing software industry and the burst of open source projects (e.g., SourceForge.net has over 180,000 registered open source projects as of November 2008), software theft (or plagiarism) has become a very serious concern to honest software companies and open source communities. As one example, in 2005 it was determined in a federal court trial that IBM should pay an independent software vendor Compuware $140 million to license its software and $260 million to purchase its services[1] because it was discovered that certain IBM products contained code from Compuware. Although some attack or theft detection approaches have been proposed, these approaches are limited to meet several highly desired requirements. For example, both attack and theft detection approaches should be resilient to code obfuscation techniques; attack detection approaches should detect new or unknown attacks; software theft detection approaches should be able to detect software component theft.

1.1 Protecting Software from Attack

Among the various types of vulnerabilities, the most serious one exploited by attackers is buffer overflow based remote code execution, which exists pervasively in Internet services (e.g., port 80 Web service), OS services (e.g., Windows DCE-RPC or CIFS/SMB), database services, applications (e.g., browser plug-ins), and so on. An attack for this vulnerability uses a network request to inject a piece of exploit code into the body of a service or application software. Once such exploit code is executed, the attacker may gain full control of the victim machine. In different attacks, exploit code may be in different forms: a piece of shellcode to break into a certain type of hosts, an infection vector for Internet worms such as CodeRed and Slammer, and so on. From both CERT [4] and Microsoft Security Bulletin [11] we can clearly see that the majority of attacks in the Microsoft Windows family are buffer-overflow based automatic remote code execution.

One approach to detect attack code inside network flows is using network intrusion detection systems (NIDSes) such as Snort [15]. NIDSes exploit manually prepared string-matching signatures to identify attacks. One advantage of this approach is that it requires no server side changes or source code. More importantly, countermeasures can be taken even before the attack begins affecting the target program. However, an limitation of string-matching signature based defenses is that they are not very resilient to *code obfuscation* such as junk instruction insertion and instruction reordering of exploit code.

To address the limitations of the signature-based defense mentioned above, several research has recently proposed to identify and analyze the attack code by program analysis approach [51, 24]. These program analysis approaches are based on the observation that remote exploits typically contain executables, whereas legitimate client requests never contain executables in most Internet services such as Web services and SQL services. Compared with string-matching signature based approaches, static analysis approaches have two notable advantages. First, they can detect new (or zero-day) exploit code for both known and unknown vulnerabilities. Second, they are more resilient to polymorphism and metamorphism. However, these existing program analysis approaches exploit special rules which can only separate code from data in a few cases. Moreover, by applying

some anti-static-analysis obfuscation techniques such as self-modifying and indirect jump[69, 57], an attacker can evade these static analysis approaches.

The goal of this dissertation is to provide a general approach which distinguishes code from data and is robust to most obfuscation techniques. To achieve this goal, several new program analysis approaches are proposed.

1.2 Protecting Software from Theft

Software theft is an action of unauthorized copy and/or use of software. There are three forms of software theft. In the software piracy case, software pirates crack software and resell them to others. In the software plagiarism case, some less self-disciplined programmers plagiarize source code from open source projects. Finally, a software company steals a portion of a program from its competitor and incorporate the portion in its own software. The focus of the dissertation is on the last two cases.

To protect software from theft, Collberg and Thoborson [28] proposed software watermark techniques. Software watermark is a unique identifier embedded in the protected software, which is hard to remove but easy to verify. However, most of commercial and open source software do not have software watermarks embedded. On the other hand, "a sufficiently determined attackers will eventually be able to defeat any watermark." [27].

A new kind of software protection technique called software birthmark were recently proposed to complement software watermark . A software birthmark is a unique characteristic that a program possesses and can be used to identify the program. Software birthmarks can be classified as static birthmarks and dynamic birthmarks. Static birthmarks rely on static characteristic of program code [80]; Dynamic birthmarks rely on the characteristic of run-time behavior of a program [74, 62, 81].

Though some initial research has been done on software birthmarks [80, 74, 62, 81], existing schemes are still limited to meet the following five highly desired requirements: Resiliency to semantics-preserving obfuscation techniques[29]; Capability to detect theft of components, which may be only a small part of the original program; Scalability to detect large-scale commercial or open source software theft;

Applicability to binary executables, because the source code of a suspected software product often cannot be obtained until some strong evidences are collected; Independence to platforms such as operating systems and program languages.

Our goal in this dissertation is to design a software theft detection system which achieve all the five requirements. To achieve this goal, a dynamic analysis approach is used and a dynamic analysis system is designed and implemented.

1.3 Summary of Contributions

The contribution of this dissertation is three-fold. First, a novel data flow analysis approach called *code abstraction* to separate code from data is presented. Unlike the previous code detection algorithms, this approach is generic, fast, and robust to code obfuscation techniques. A system called SigFree based on this approach is designed and implemented.

Second, a generic defense named STILL is proposed based on *S*tatic *T*aint and *I*nitia*L*ization ana*L*yses. STILL is the first one that can detect self-modifying code and indirect jump code, and the most comprehensive static analysis solution in defending against anti-signature, anti-static-analysis and anti-emulation code obfuscation. The experiment results show that STILL achieves high accuracy in detecting the various types of exploit code, no matter they are obfuscated or not.

Third, a system call dependence graph based software birthmark called *SCDG birthmark* is proposed. A dynamic analysis tool is designed and implemented for generating system call traces and SCDGs. To our knowledge, our detection system based on SCDG birthmark is the first one that is capable of software *component* theft detection where only partial code is stolen. The strength of these birthmarks are demonstrated against various evasion techniques, including those based on different compilers and different compiler optimization levels as well as obfuscation tools. Unlike the existing work that were evaluated through small or toy software, a set of large software are evaluated.

1.4 Organization

The remaining of the thesis is organized as follows. Chapter 2 reviews related work. Chapter 3 presents a novel static program analysis method and a signature-free system based on the new method. Chapter 4 focuses on another defense system which is robust to advanced obfuscation techniques. Chapter 5 presents a system call dependence graph birthmark based software theft detection system. We conclude the dissertation in Chapter 6.

Chapter 2

Related Work

This dissertation is mainly relevant to the following work.

2.1 Detecting Attack

2.1.1 Attack Detection by Signature

Automatic signature generation techniques were proposed to automatically extract string-matching signatures from malicious payloads to contain zero-day attacks [76, 65, 88]. However, a common limitation of string-matching signature based defenses is that they or their flow classifiers are not very resilient to polymorphic and metamorphic obfuscation, especially when under time pressure.

Recently, some researchers proposed several semantic-aware signatures by static or dynamic analysis techniques. Vigilante [32] and TaintCheck [66] exploit dynamic data flow and taint analysis techniques detect unknown and generate signatures. Covers [56] exploit post-crash symptom diagnosis extracts the signatures. Liang and Sekar proposed ARBOR [55] which can automatically generate vulnerability-oriented signatures by identifying characteristic features of attacks and using program context. Brumley et al. [23, 22] firstly propose techniques for automatically create vulnerability signatures for software. Semantic-aware signatures can block both the attack messages that contain code and the attack messages that do not contain any code, but they need the signatures to be firstly generated. Moreover, they either suffer from significant run-time overhead or need special analysis or

diagnosis facilities which are not commonly available in commercial services.

Wang and Stolfo [86] proposed PAYL which use anomaly detection on packet payload to detect worms and generate signature. PAYL is first trained with normal network flow traffic and then uses some byte-level statistical measures to detect exploit code. The anomaly detection techniques could be evaded by statistically mimics normal traffic [48].

2.1.2 Attack Detection by Static Analysis

Kruegel et al. [51] exploited control flow structures to detect polymorphic shellcode and worms. Chinchani and Berg [24] used pattern-matching and data flow analysis techniques to detect exploit code inside network flows. SigFree[87] blocks attacks by detecting the presence of code in data requests. It uses a new technique called code abstraction to differentiate code and data. One benefit of these static analysis approaches is that they can detect both foreseen exploit code exploiting known vulnerabilities and zero-day exploit code exploiting unknown vulnerabilities. In addition, they are in general more resilient to polymorphism and metamorphism (than string-matching signatures). However, Polychronakis et al. [69] demonstrated that some anti-static-analysis techniques such as self-modifying and indirect jump can easily thwart these existing static analysis techniques.

2.1.3 Attack Detection by Emulation

Polychronakis et al. [69] firstly proposed a NIDS-embedded CPU emulator to detect polymorphic shellcode. Zhang et al. [91] proposed another emulator to detect polymorphic shellcode. In addition, they use a static analysis method to identify the start point of polymorphic shellcode which is potentially faster than [69]. The emulators, being a dynamic analyzer, are immune to most anti-static-analysis techniques. However, dynamic analysis is vulnerable to several anti-emulation techniques, which have existed in virus writer community for many years [79]. For example, it may be thwarted by random worms that only execute on a specific condition such as a randomly generated date or time. Another example is that it must use some heuristics to determine when to stop analyzing a program, because execution may not terminate. An experienced attacker may bypass the stopping

heuristics by introducing a delay loop that simply wastes cycles. We note that the emulators in [69, 91] detect only polymorphic shellcode, thus plain (with or without metamorphism) exploit code such as worm Code Red will evade its detection.

2.2 Detecting Theft

We roughly group the literature of software theft detection into two categories: software birthmark and clone detection.

2.2.1 Software Birthmark

There are four classes of software birthmark: static source code based birthmark [80], static executable code based birthmark [63], dynamic WPP based birthmark [62], and dynamic API based birthmark [74, 81].

Static source code based birthmark: Tamada[80] *et al.* proposed four types of static birthmark: Constant Values in Field Variables Birthmark (CVFV), Sequence of Method Calls Birthmark (SMC), Inheritance Structure Birthmark (IS) and Used Classes Birthmark (UC). All of the four types are vulnerable to obfuscation techniques mentioned in [63]. In addition, they need to access source code and only work for object-oriented programming language.

Static executable code based birthmark: Myles and Collberg [63] proposed a opcode-level k-gram based static birthmark. Opcode sequences of length k are extracted from a program and k-gram techniques which were used to detect similarity of documents are exploited to the opcode sequences. Although the k-gram static birthmark is more robust than Tamadas birthmark, it is still strongly vulnerable to some well-known obfuscations such as statement reordering, junk instruction insertion and other semantic-preserved transformation techniques such as compiler optimization.

Dynamic WPP based birthmark: Myles and Collberg [62] proposed a whole program path (WPP) based dynamic birthmark. WPP is originally used to represent the dynamic control flow of a program. WPP birthmark is robust to some control flow obfuscations such as opaque prediction, but is still vulnerable to many semantic-preserving transformation such as loop unwinding. Moreover, WPP

birthmark may not work for large-scale programs due to overwhelming volume of WPP traces.

Dynamic API based birthmark: Tamada et al. [81, 82] also introduced two types of dynamic birthmark for Windows applications: Sequence of API Function Calls Birthmark (EXESEQ) and Frequency of API Function Calls Birthmark (EXEFREQ). In EXESEQ, the sequence of Windows API calls are recorded during the execution of a program. These sequences are directly compared to find similarity. In EXEFREQ, the frequency of each Windows API calls are recorded during the execution of a program. The frequency distribution is used as the birthmark. Schuler et al. [74] proposed a dynamic birthmark for Java. The call sequences to Java standard API are recorded and the short sequences at object level are used as a birthmark. Their experiments showed that API birthmarks are more robust to obfuscation than WPP birthmark in their evaluation. Unlike the Java or Windows API based birthmarks that are platform dependent, system call birthmarks can be used on any platform. In addition, system call birthmarks are more robust to counter-attacks than API-based ones. To evade API-based birthmarks, attackers may hide API calls by embedding their own implementation of some API routines. However, there are no easy ways to replace "system calls" without recompiling the kernel because because system call is the only way to gain privilege in modern operating systems. More importantly, existing API-based birthmarks have not been evaluated to protect core components theft.

2.2.2 Clone Detection

A close research field to software birthmark is clone detection. Clone detection is a technique to find the duplicate code ("clones") in a large-scale program. Existing techniques for clone detection can be classified as four categories: String-based [17], AST-based [20, 49], Token-based[43, 71, 73] and PDG-based [58, 50]. *String-based:* Each line of source code is considered as a string and the whole program is considered as a sequence of strings. A code fragment is labelled as clone if the corresponding sequence of strings is the same as another code fragment from original program. *AST-based:* The abstract syntax trees (AST) are extracted from programs by analyzing their syntax. Then the ASTs are directly compared.

If there are common subtrees, clone may exist. *Token-based:* A program is first parsed to a sequence of tokens. The sequences of tokens are compared to find clone. *PDG-based:* A program dependency graph is a graph which represents the control flow and data flow relations between the statements in a program procedure. To find clone, two PDGs are extracted from two programs (by some static analysis tools) and compared to find relaxed subgraph isomorphism.

Besides to be used to decrease code size and facilitate maintenance, clone detection can be also be used to detect software plagiarism. However, existing clone detection techniques are not robust to code obfuscation. String-based schemes are fragile even by simply renaming identifiers in programs. AST-based schemes are resilient to identifier renaming, but weak against statement reordering and control replacement. Token-based schemes are resilient to identifier renaming, but weak against junk code insertion and statement reordering. Because PDGs contain semantic information of programs, PDG-based schemes are more robust than the other three types of existing schemes. However, PDG-based is still vulnerable to many semantics-preserving transformations such as control flow flattening and opaque predicates. Moreover, all clone detection techniques need to access source code.

Chapter 3

SigFree: A Signature-free Buffer Overflow Attack Blocker

3.1 Introduction

Throughout the history of cyber security, buffer overflow is one of the most serious vulnerabilities in computer systems. Buffer overflow vulnerability is a root cause for most of the cyber attacks such as server breaking-in, worms, zombies, and botnets. A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and, depending on what is stored there, the behavior of the program itself might be affected [53]. Although taking a broader viewpoint, buffer overflow attacks do not always carry binary code in the attacking requests (or packets) [1] , code-injection buffer overflow attacks such as stack smashing probably count for most of the buffer overflow attacks that have happened in the real world.

Although tons of research has been done to tackle buffer overflow attacks, existing defenses are still quite limited in meeting four highly-desired requirements: (R1) *simplicity* in maintenance; (R2) *transparency* to existing (legacy) server OS, application software, and hardware; (R3) *resiliency* to obfuscation; (R4) economical Internet wide deployment. As a result, although several very secure solutions

[1] A buffer overflow attack may corrupt control-flow or data without injecting code such as return-to-libc attacks and data-pointer modification [68]. In this dissertation, we only focus on code-injection buffer overflow attacks.

have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to succeed on a daily basis.

To see how existing defenses are limited in meeting these four requirements, let us break down the existing buffer overflow defenses into six classes: (1A) Finding bugs in source code. (1B) Compiler extensions. (1C) OS modifications. (1D) Hardware modifications. (1E) Defense-side obfuscation [44, 19]. (1F) Capturing code running symptoms of buffer overflow attacks [66, 32, 56, 89]. (Note that the above list does not include binary code analysis based defenses which we will address shortly.) We may briefly summarize the limitations of these defenses in terms of the four requirements as follows. (a) Class 1B, 1C, 1D, and 1E defenses may cause substantial changes to existing (legacy) server OSes, application software, and hardware, thus they are not transparent. Moreover, Class 1E defenses generally cause processes to be terminated. As a result, many businesses do not view these changes and the process termination overhead as economical deployment. (b) Class 1F defenses can be very secure, but they either suffer from significant run-time overhead or need special auditing or diagnosis facilities which are not commonly available in commercial services. As a result, Class 1F defenses have limited transparency and potential for economical deployment. (c) Class 1A defenses need source code, but source code is unavailable to many legacy applications.

Besides buffer overflow defenses, worm signatures can be generated and used to block buffer overflow attack packets [76, 45, 65]. Nevertheless, they are also limited in meeting the four requirements, since they either rely on signatures, which introduce maintenance overhead, or are not very resilient to attack-side obfuscation.

To overcome the above limitations, in this chapter we propose SigFree, a online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that "the nature of communication to and from network services is predominantly or exclusively data and not executable code." [24]. In particular, as summarized in [24], (a) on Windows platforms, most web servers (port 80) accept data only; remote access services (ports 111, 137, 138, 139) accept data only; Microsoft SQL Servers (port 1434), which are used to monitor Microsoft SQL Databases, accept data only. (b) On Linux platforms, most Apache web servers (port 80) accept data only; BIND (port 53) accepts data

only; SNMP (port 161) accepts data only; most Mail Transport (port 25) accepts data only; Database servers (Oracle, MySQL, PostgreSQL) at ports 1521, 3306 and 5432 accept data only.

Since remote exploits are typically binary executable code, this observation indicates that if we can precisely distinguish (service requesting) messages containing binary code from those containing no binary code, we can protect most Internet services (which accept data only) from code-injection buffer overflow attacks by blocking the messages that contain binary code.

Accordingly, SigFree (Figure 3.1) works as follows. SigFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree first uses a new $O(N)$ algorithm, where N is the byte length of the message, to disassemble and distill all possible instruction sequences from the message's payload, where every byte in the payload is considered as a possible starting point of the code embedded (if any). However, in this phase some data bytes may be mistakenly decoded as instructions. In phase 2, SigFree uses a novel technique called *code abstraction*. Code abstraction first uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions (Scheme 2) or *dependence degree* (Scheme 3) to a threshold to determine if this instruction sequence (distilled in phase 1) contains code. Unlike the existing code detection algorithms [83, 51, 24] that are based on signatures, rules, or control flow detection, SigFree is generic and hard for exploit code to evade.

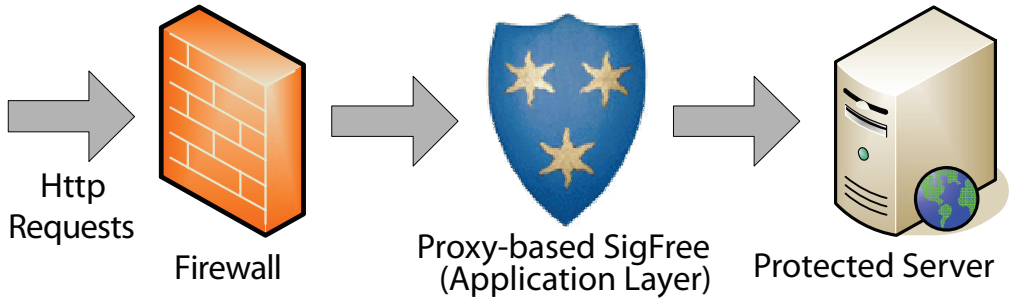


Figure 3.1. SigFree is an application layer blocker between the protected server and the corresponding firewall.

We have implemented a SigFree prototype as a proxy to protect web servers. Our empirical study shows that there exists clean-cut "boundaries" between code-

embedded payloads and data payloads when our code-data separation criteria are applied. We have identified the "boundaries" (or thresholds) and been able to detect/block all 50 attack packets generated by Metasploit framework [10], all 700 polymorphic shellcode packets generated by polymorphic shellcode engines Countdown [10], JumpCallAdditive [10], JempiScodes [8], ADMmutate [59] and CLET [34], and worm Slammer, CodeRed and a CodeRed variation, when they are well mixed with various types of data packets. Also, our experiment results show that the extra processing delay caused by SigFree to client requests is negligible.

The merits of SigFree are summarized below. They show that SigFree has taken a main step forward in meeting the four requirements aforementioned.

⊙ *SigFree is signature free, thus it can block new and unknown buffer overflow attacks.*

⊙ Without relying on string-matching, *SigFree is immunized from most attack-side obfuscation methods.*

⊙ *SigFree uses generic code-data separation criteria instead of limited rules.* This feature separates SigFree from [24], an independent work that tries to detect code-embedded packets.

⊙ *Transparency.* SigFree is an out-of-the-box solution that requires no server side changes.

⊙ *SigFree is an economical deployment with very low maintenance cost*, which can be well justified by the aforementioned features.

The rest of the chapter is organized as follows. In Section 3.2, we give an overview of SigFree. In Sections 3.3 and 3.4, we introduce the instruction sequence distiller component and the instruction sequence analyzer component of SigFree, respectively. In Section 3.5, we show our experimental results. Finally, we discuss some remaining research issues in Section 3.6 and conclude the paper in Section 3.7.

3.2 SigFree Overview

3.2.1 Basic Definitions and Notations

This section provides the definitions that will be used in the rest of the paper.

Definition 1. *(instruction sequence) An instruction sequence is a sequence of*

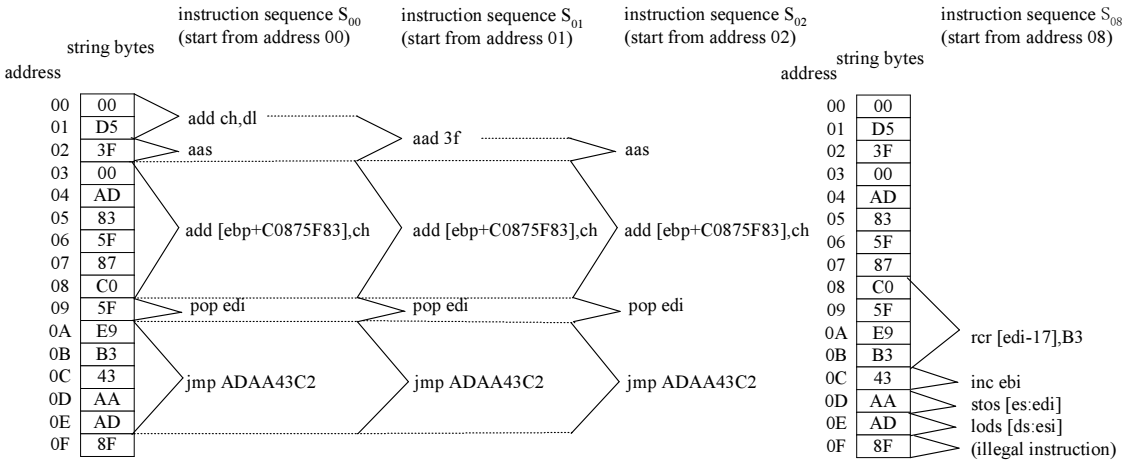


Figure 3.2. Instruction sequences distilled from a substring of a GIF file. We assign an address to every byte of the string. Instruction sequences s_{00}, s_{01}, s_{02} and s_{08} are distilled by disassembling the string from addresses 00, 01, 02 and 08, respectively.

CPU instructions which has one and only one entry instruction and there exist at least one execution path from the entry instruction to any other instruction in this sequence.

A fragment of a program in machine language is an instruction sequence, but an instruction sequence is not necessarily a fragment of a program. In fact, we may distill instruction sequences from any binary strings. This poses the fundamental challenge to our research goal. Figure 3.2 shows four instruction sequences distilled from a substring of a GIF file. Each instruction sequence is denoted as s_i in Figure 3.2, where i is the entry location of the instruction sequence in the string. These four instruction sequences are not fragments of a real program, although they may also be executed in a specific CPU. Below we call them *random instruction sequences*, whereas use the term *binary executable code* to refer to a fragment of a real program in machine language.

Definition 2. *(instruction flow graph) An instruction flow graph (IFG) is a directed graph $G = (V, E)$ where each node $v \in V$ corresponds to an instruction and each edge $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from instruction v_i to instruction v_j.*

Unlike traditional control flow graph (CFG), a node of an IFG corresponds to a single instruction rather than a basic block of instructions. To completely model

the control flow of an instruction sequence, we further extend the above definition.

Definition 3. *(extended instruction flow graph) An extended instruction flow graph (EIFG) is a directed graph $G = (V, E)$ which satisfies the following properties: each node $v \in V$ corresponds to an instruction, an illegal instruction (an "instruction" which cannot be recognized by CPU), or an external address (a location which is beyond the address scope of all instructions in this graph); each edge $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from instruction v_i to instruction v_j, to illegal instruction v_j, or to an external address v_j.*

Accordingly, we name the types of nodes in an EIFG *instruction node*, *illegal instruction node*, and *external address node*.

The reason that we define IFG and EIFG is to model two special cases which CFG cannot model (the difference will be very evident in the following sections). First, in an instruction sequence, control may be transferred from an instruction node to an illegal instruction node. For example, in instruction sequence s_{08} in Figure 3.2, the transfer of control is from instruction "lods [ds:esi]" to an illegal instruction at address $0F$. Second, control may be transferred from an instruction node to an external address node. For example, instruction sequence s_{00} in Figure 3.2 has an instruction "jmp ADAAC3C2", which jumps to external address ADAAC3C2.

3.2.2 Assumptions

In this paper, we focus on buffer overflow attacks whose payloads contain executable code in machine language, and we assume normal requests do not contain executable machine code. A normal request may contain any data, parameters, scripts or even a SQL statement. Note that although SQL statements and scripts are executable in the application level, they cannot be executed directly by a CPU. As such, SQL statements and scripts are not viewed as executable in our model. Application level attacks such as data manipulation and SQL injection are out of the scope.

Though SigFree is a generic technique which can be applied to any instruction set, for concreteness we assume the web server runs the Intel IA32 instruction set, the most popular instruction set running inside a web server today.

3.3 Instruction Sequence Distiller

This section first describes an effective algorithm to distill instruction sequences from requests, followed by several pruning techniques to reduce the processing overhead of instruction sequence analyzer.

3.3.1 Distilling Instruction Sequences

To distill an instruction sequence, we first assign an address (starting from zero) to every byte of a request, where address is an identifier for each location in the request. Then, we disassemble the request from a certain address until the end of the request is reached or an illegal instruction opcode is encountered. There are two traditional disassembly algorithms: *linear sweep* and *recursive traversal*[75, 57]. The linear sweep algorithm begins disassembly at a certain address, and proceeds by decoding each encountered instruction. The recursive traversal algorithm also begins disassembly at a certain address, but it follows the control flow of instructions.

In this paper, we employ the recursive traversal algorithm, because it can obtain the control flow information during the disassembly process. Intuitively, to get all possible instruction sequences from a N-byte request, we simply execute the disassembly algorithm N times and each time we start from a different address in the request. This gives us a set of instruction sequences. The running time of this algorithm is $O(N^2)$.

One drawback of the above algorithm is that the same instructions are decoded many times. For example, instruction "pop edi" in Figure 3.2 is decoded many times by this algorithm. To reduce the running time, we design a memorization algorithm [31] by using a data structure, which is an EIFG defined earlier, to represent the instruction sequences. To distill all possible instruction sequences from a request is simply to create the EIFG for the request. An EIFG is used to represent all possible transfers of control among these instructions. In addition, we use an instruction array to represent all possible instructions in a request. To traverse an instruction sequence, we simply traverse the EIFG from the entry instruction of the instruction sequence and fetch the corresponding instructions from the instruction array. Figure 3.3 shows the data structure for the request

shown in Figure 3.2. The details of the algorithm for creating the data structure are described in Algorithm 1. Clearly, the running time of this algorithm is $O(N)$, which is optimal as each address is traversed only once.

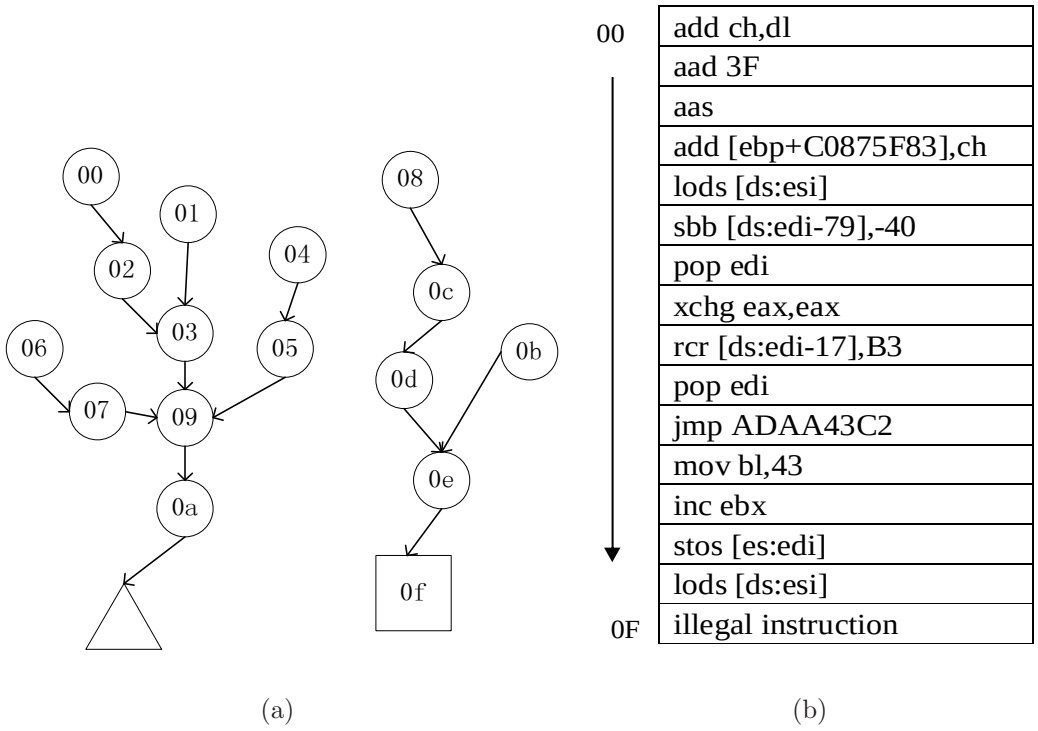


Figure 3.3. Data structure for the instruction sequences distilled from the request in Figure 3.2. (a) Extended instruction flow graph. Circles represent instruction nodes; triangles represent external addresses; rectangles represent illegal instructions. (b) The array of all possible instructions in the request.

3.3.2 Excluding Instruction Sequences

The previous step may output many instruction sequences at different entry points. Next we exclude some of them based on several heuristics. Here *excluding an instruction sequence means that the entry of this sequence is not considered as the real entry for the embedded code (if any).*

The fundamental rule in excluding instruction sequences is not to affect the decision whether a request contains code or not. This rule can be translated into the following technical requirements: if a request contains a fragment of a program,

Algorithm 1 Distill all instruction sequences from a request

```
initialize EISG G and instruction array A to empty
for each address i of the request do
  add instruction node i to G
end for
i ← the start address of the request
while i <= the end address of the request do
  inst ← decode an instruction at i
  if inst is illegal then
    A[i] ← illegal instruction inst
    set type of node i "illegal node" in G
  else
    A[i] ← instruction inst
    if inst is a control transfer instruction then
      for each possible target t of inst do
        if target t is an external address then
          add external address node t to G
        end if
        add edge e(node i, node t) to G
      end for
    else
      add edge e(node i, node i + inst.length) to G
    end if
    i ← i + 1
  end if
end while
```

the fragment must be one of the remaining instruction sequences or a subsequence of a remaining instruction sequence, or it differs from a remaining sequence only by few instructions.

Step 1 If instruction sequence s_a is a subsequence of instruction sequence s_b, we exclude s_a. The rationale for excluding s_a is that if s_a satisfies some characteristics of programs, s_b also satisfies these characteristics with a high probability.

This step helps exclude lots of instruction sequences since many distilled instruction sequences are subsequences of the other distilled instruction sequences. For example, in Figure 3.3(a), instruction sequence s_{02}, which is a subsequence of instruction sequence s_{00}, can be excluded. Note that here we only exclude instruction sequence s_{02} rather than remove node v_{02}. Similarly, instruction sequences

s_{03},s_{05}, s_{07}, s_{09},s_{0a},s_{0c},s_{0d} and s_{0e} can be excluded.

Step 2 If instruction sequence s_a merges to instruction sequence s_b after a few instructions (e.g., 4 in our experiments) and s_a is no longer than s_b, we exclude s_a. It is reasonable to expect that s_b will preserve s_a's characteristics.

Many distilled instruction sequences are observed to merge to other instructions sequences after a few instructions. This property is called self-repairing[57] in Intel IA-32 architecture. For example, in Figure 3.3(a) instruction sequence s_{01} merges to instruction sequence s_{00} only after one instruction. Therefore, s_{01} is excluded. Similarly, instruction sequences s_{04}, s_{06} and s_{0b} can be excluded.

Step 3 For some instruction sequences, when they are executed, whichever execution path is taken, an illegal instruction is *inevitably reached*. We say an instruction is inevitably reached if two conditions holds. One is that there are no cycles (loops) in the EIFG of the instruction sequence; the other is that there are no external address nodes in the EIFG of the instruction sequence.

We exclude the instruction sequences in which illegal instructions are inevitably reached, because causing the server to execute an illegal instruction is not the purpose of a buffer overflow attack (this assumption was also made by others [51, 24], implicitly or explicitly). Note that however the existence of illegal instruction nodes cannot always be used as a criteria to exclude an instruction sequence unless they are inevitably reached; otherwise attackers may obfuscate their program by adding *non-reachable* illegal instructions.

Based on this heuristic, we can exclude instruction sequence s_{08} in Figure 3.3(a), since it will eventually execute an illegal instruction v_{0f}.

After these three steps, in Figure 3.3(a) only instruction sequence s_{00} is left for consideration in the next stage.

3.4 Instruction Sequences Analyzer

A distilled instruction sequence may be a sequence of random instructions or a fragment of a program in machine language. In this section, we propose three schemes to differentiate these two cases. Scheme 1 exploits the operating system characteristics of a program; Scheme 2 and Scheme 3 exploit the data flow characteristics of a program. Scheme 1 is slightly faster than Scheme 2 and Scheme 3,

whereas Scheme 2 and Scheme 3 are much more robust to obfuscation.

3.4.1 Scheme 1

A program in machine language is dedicated to a specific operating system; hence, a program has certain characteristics implying the operating system on which it is running, for example calls to operating system or kernel library. A random instruction sequence does not carry this kind of characteristics. By identifying the call pattern in an instruction sequence, we can effectively differentiate a real program from a random instruction sequence.

More specifically, instructions such as "call" and "int 0x2eh" in Windows and "int 0x80h" in Linux may indicate system calls or function calls. However, since the op-codes of these call instructions are only one byte, even normal requests may contain plenty of these byte values. Therefore, using the number of these instructions as a criteria will cause a high false positive rate. To address this issue, we use a pattern composed of several instructions rather than a single instruction. It is observed that before these call instructions there are normally one or several instructions used to transfer parameters. For example, a "push" instruction is used to transfer parameters for a "call" instruction; some instructions that set values to registers al, ah, ax, or eax are used to transfer parameters for "int" instructions. These call patterns are very common in a fragment of a real program. Our experiments in Section 3.5 show that by selecting the appropriate parameters we can rather accurately tell whether an instruction sequence is an executable code or not.

Scheme 1 is fast since it does not need to fully disassemble a request. For most instructions, we only need to know their types. This saves lots of time in decoding operands of instructions.

Note that although Scheme 1 is good at detecting most of the known buffer overflow attacks, it is vulnerable to obfuscation. One possible obfuscation is that attackers may use other instructions to replace the "call" and "push" instructions. Figure 3.4 shows an example of obfuscation, where "call eax" instruction is substituted by "push J4" and "jmp eax". Although we cannot fully solve this problem, by recording this kind of instruction replacement patterns, we may still be able to

detect this type of obfuscation to some extent.

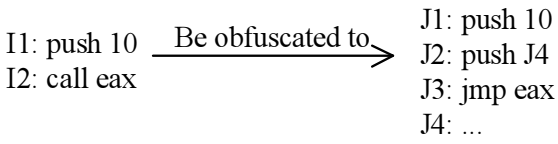


Figure 3.4. An obfuscation example. Instruction "call eax" is substituted by "push J4" and "jmp eax".

3.4.2 Scheme 2

Next we propose Scheme 2 to detect the aforementioned obfuscated buffer overflow attacks. Scheme 2 exploits the data flow characteristics of a program. Normally, a random instruction sequence is full of data flow anomalies, whereas a real program has few or no data flow anomalies. However, the number of data flow anomalies cannot be directly used to distinguish a program from a random instruction sequence because an attacker may easily obfuscate his program by introducing enough data flow anomalies.

In this paper, we use the detection of data flow anomaly in a different way called *code abstraction*. We observe that when there are data flow anomalies in an execution path of an instruction sequence, some instructions are useless, whereas in a real program at least one execution path has a certain number of useful instructions. Therefore, if the number of useful instructions in an execution path exceeds a threshold, we conclude the instruction sequence is a segment of a program.

Data Flow Anomaly The term data flow anomaly was originally used to analyze programs written in higher level languages in the software reliability and testing field[36, 41]. In this paper, we borrow this term and several other ones to analyze instruction sequences.

During a program execution, an instruction may impact a variable (register, memory location or stack) on three different ways: *define*, *reference*, and *undefine*. A variable is defined when it is set a value; it is referenced when its value is referred to; it is undefined when its value is not set or set by another undefined variable. Note that here the definition of undefined is different from that in a high

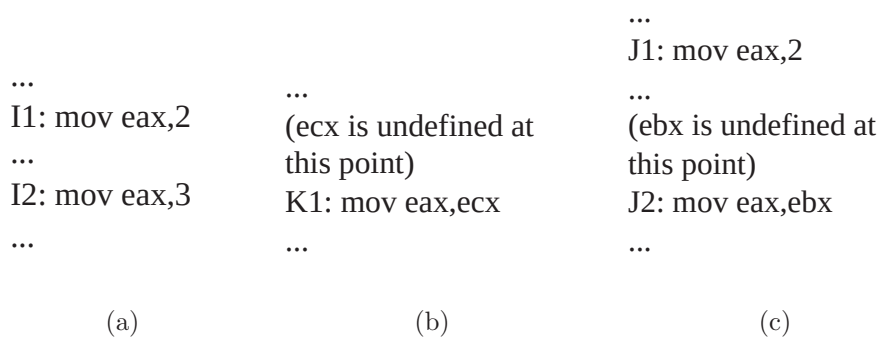


Figure 3.5. Data flow anomaly in execution paths. (a) define-define anomaly. Register eax is defined at I1 and then defined again at I2. (b) undefine-reference anomaly. Register ecx is undefined before K1 and referenced at K1 (c) define-undefine anomaly. Register eax is defined at J1 and then undefined at J2.

level language. For example, in a C program, a local variable of a block becomes undefined when control leaves the block.

A data flow anomaly is caused by an improper sequence of actions performed on a variable. There are three data flow anomalies: *define-define*, *define-undefine*, and *undefine-reference*[41]. The define-define anomaly means that a variable was defined and is defined again, but it has never been referenced between these two actions. The undefine-reference anomaly indicates that a variable that was undefined receives a reference action. The define-undefine anomaly means that a variable was defined, and before it is used it is undefined. Figure 3.5 shows an example.

Detection of Data Flow Anomalies There are static[36] or dynamic[41] methods to detect data flow anomalies in the software reliability and testing field. Static methods are not suitable in our case due to its slow speed; dynamic methods are not suitable either due to the need for real execution of a program with some inputs. As such, we propose a new method called code abstraction, which does not require real execution of code. As a result of the code abstraction of an instruction, a variable could be in one of the six possible states. The six possible states are state U: undefined; state D: defined but not referenced; state R: defined and referenced; state DD : abnormal state define-define; state UR: abnormal state undefine-reference; and state DU: abnormal state define-undefine. Figure 3.6 depicts the state diagram of these states. Each edge in this state diagram is

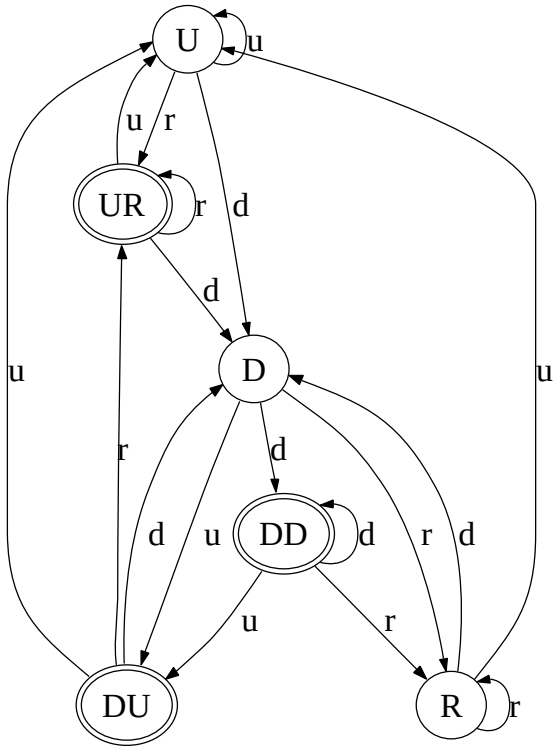


Figure 3.6. State diagram of a variable. State U: undefined, state D: defined but not referenced, state R: defined and referenced, state DD : abnormal state define-define, state UR: abnormal state undefine-reference and state DU: abnormal state define-undefine.

associated with d, r, or u, which represents "define", "reference", and "undefine", respectively.

We assume that a variable is in "undefined" state at the beginning of an execution path. Now we start to traverse this execution path. If the entry instruction of the execution path defines this variable, it will enter the state "defined". Then, it will enter another state according to the next instruction, as shown in Figure 3.6. Once the variable enters an abnormal state, a data flow anomaly is detected. We continue this traversal to the end of the execution path. This process enables us to find all the data flow anomalies in this execution path.

Pruning Useless Instructions Next we leverage the detected data flow anomalies to remove useless instructions. A *useless* instruction of an execution path is an instruction which does not affect the results of the execution path; otherwise, it is called *useful* instructions. We may find a useless instruction from

a data flow anomaly. When there is an undefine-reference anomaly in an execution path, the instruction which causes the "reference" is a useless instruction. For instance, the instruction $K1$ in Figure 3.5, which causes undefine-reference anomaly, is a useless instruction. When there is a define-define or define-undefine anomaly, the instruction that caused the former "define" is also considered as a useless instruction. For instance, the instructions $I1$ and $J1$ in Figure 3.5 are useless instructions because they caused the former "define" in either the define-define or the define-undefine anomaly.

After pruning the useless instructions from an execution path, we will get a set of useful instructions. If the number of useful instructions in an execution path exceeds a threshold, we will conclude the instruction sequence is a segment of a program. Algorithm 2 shows our algorithm to check if the number of useful instructions in an execution path exceeds a threshold. The algorithm involves a search over an EISG in which the nodes are visited in a specific order derived from a depth first search. The algorithm assumes that an EISG G and the entry instruction of the instruction sequence are given, and a push down stack is available for storage. During the search process, the visited node (instruction) is abstractly executed to update the states of variables, find data flow anomaly, and prune useless instructions in an execution path.

Handling Special Cases Next we discuss several special cases in the implementation of Scheme 2.

General purpose instruction The instructions in the IA32 instruction set can be roughly divided into four groups: general purpose instructions, floating point unit instructions, extension instructions, and system instructions. General purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operation, which are commonly used by programmers to write applications and system software that run on IA-32 processors[7]. General purpose instructions are also the most often used instructions in malicious code. We believe that malicious codes must contain a certain number of general purpose instructions to achieve the attacking goals. Other types of instructions may be leveraged by an attacker to obfuscate his real-purpose code, e.g., used as garbage in garbage insertion. As such, we consider other groups of instructions as useless instructions.

Initial state of registers For registers, we set their initial states to "undefined"

Algorithm 2 check if the number of useful instructions in an execution path exceeds a threshold

Input: *entry* instruction of an instruction sequence, EISG G

```
total ← 0; useless ← 0 ; stack ← empty
initialize the states of all variables to "undefined"
push the entry instruction,states,total and useless to stack
while stack is not empty do
  pop the top item of stack to i,states,total and useless
  if total − useless greater than a threshold then
    return true
  end if
  if i is visited then
    continue ( passes control to the next iteration of the WHILE loop)
  end if
  mark i visited
  total ← total + 1
  Abstractly execute instruction i (change the states of variables according to
  instruction i)
  if there is a define-define or define-undefine anomaly then
    useless ← useless + 1
  end if
  if there is a undefine-reference anomaly then
    useless ← useless + 1
  end if
  for each instruction j directly following i in the G do
    push j, states ,total and useless to stack
  end for
end while
return false
```

at the beginning of an execution path. The register "esp", however, is an exception since it is used to hold the stack pointer. Thus, we set register esp "defined" at the beginning of an execution path.

Indirect address An indirect address is an address that serves as a reference point instead of an address to the direct memory location. For example, in the instruction "move eax,[ebx+01e8]", register "ebx" may contain the actual address of the operand. However, it is difficult to know the run-time value of register "ebx". Thus, we always treat a memory location to which an indirect address points as

state "defined" and hence no data flow anomaly will be generated. Indeed, this treatment successfully prevents an attacker from obfuscating his code using indirect addresses.

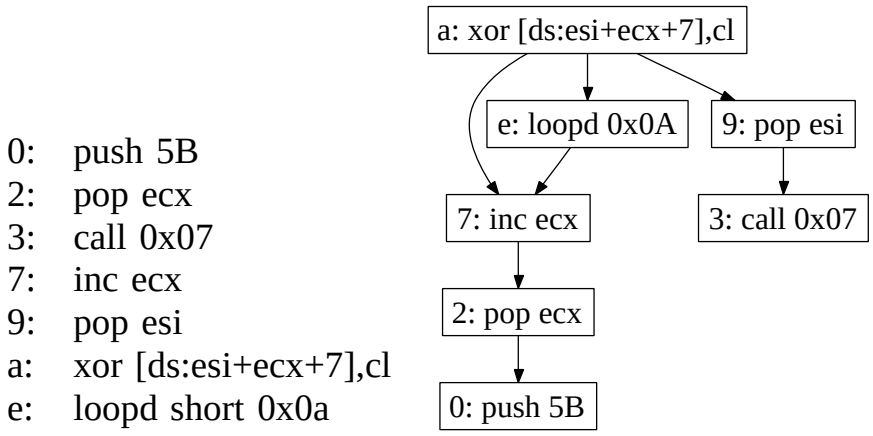
Useless control transfer instructions A CTI (Control Transfer Instruction) is useless if the corresponding control condition is undefined or the control transfer target is undefined. The status flags of the EFLAGS register is used as control condition in IA-32 architecture. These status flags indicate the results of arithmetic and shift instructions, such as the ADD, SUB and SHL instructions. Condition instructions Jcc (jump on condition code cc) and LOOPcc use one or more of the status flags as condition codes and test them for branch or end-loop conditions. During a program execution at runtime, an instruction may affect a status flag on three different ways: *set*, *unset* or *undefine* [7]. We consider both *set* and *unset* are defined in code abstraction. As a result, a status flag could be in one of the two possible states - defined or undefined in code abstraction. To detect and prune useless CTIs, we also assume that the initial states of all status flags are undefined. The states of status flags are updated during the process of code abstraction.

The objective in selecting the threshold is to achieve both low false positives and low false negatives. Our experimental results in Section 3.5 show that we can achieve this objective over our test datasets. However, it is possible that attackers evade the detection by using specially-crafted code, if they know our threshold. For example, Figure 3.7(a) shows a decryption routine has only 7 useful instructions which is less than our threshold (15 to 17 in our experiments) of Scheme 2.

3.4.3 Scheme 3

We propose Scheme 3 for detecting the aforementioned specially-crafted code. Scheme 3 also exploits *code abstraction* to prune useless instructions in an instruction sequence. Unlike Scheme 2, which compares the number of useful instructions with a threshold, Scheme 3 first calculates the *dependence degree* of every instruction in the instruction sequence. If the dependence degree of any useful instructions in an instruction sequence exceeds a threshold, we conclude that the instruction sequence is a segment of a program.

Dependency is a binary relation over instructions in an instruction sequence.



(a) (b)

Figure 3.7. (a) A decryption routine which only has 7 useful instructions (b) a def-use graph of the decryption routine. Instruction a can reach all other instructions through paths $a \rightarrow e \rightarrow 7 \rightarrow 2 \rightarrow 0$ and $a \rightarrow 9 \rightarrow 3$ and therefore the dependence degree of instruction a is 6.

We say instruction j *depends* on instruction i if instruction i produces a result directly or indirectly used by instruction j. Dependency relation is transitive, that is, if i depends on j and j depends on k, then i depends on k. For example, instruction 2 directly depends on instruction 0 in Figure 3.7(a) and instruction 7 directly depends on instruction 2, and by transitive property instruction 7 depends on instruction 0. We call the number of instructions, which an instruction depends on, the *dependence degree* of the instruction.

To calculate the dependence degree of an instruction, we construct a *def-use graph*. A *def-use graph* is a directed graph $G = (V, E)$ where each node $v \in V$ corresponds to an instruction and each edge $e = (v_i, v_j) \in E$ indicates that instruction v_j produces a result directly used by instruction v_i. Obviously, the number of instructions which an instruction can reach through any path in the def-use graph is the dependence degree of the instruction. For example, Instruction a can reach all other instructions through paths $(a, e, 7, 2, 0)$ and $(a, 9, 3)$ in the def-use graph of Figure 3.7(b); therefore, the dependence degree of instruction a is 6.

We observe that even though the attackers may use only a few useful instructions, at least one useful instruction depends on a certain number of other useful

instructions in a program, whereas all useful instructions in a random instruction depend on few other useful instructions. Therefore, if the number of useful instructions an useful instruction depends on exceeds a threshold in an instruction sequence, we conclude that there are real code embedded in the request. We will show that Scheme 3 provides a higher detection rate than Scheme 2.

3.5 Experiments

In this section, we first tune the parameter (detection threshold) for each SigFree scheme based on some training data, then evaluate and compare the performance of these schemes in checking messages collected from various sources. Performance overhead of SigFree is also evaluated when deploying it to protect a web server.

3.5.1 Parameter Tuning

All three schemes use a threshold value to determine if a request contains code or not. Clearly, it is critical to set the threshold values appropriately so as to minimize both detection false positive rate and false negative rate. To find out the appropriate thresholds, we tested these schemes against 50 unencrypted attack requests generated by Metasploit framework, worm Slammer, CodeRed (CodeRed.a) and a CodeRed variation (CodeRed.c), and 1500 binary HTTP replies (52 encrypted data, 23 audio, 195 jpeg, 32 png, 1153 gif and 45 flash) intercepted on our department network. Here we choose HTTP replies rather than requests as normal data for parameter tuning, because HTTP replies contain more binaries (our test over real traces of web requests is reported in Section 3.5.3). This allows us to obtain a better threshold, which can be used to protect not only web servers but also other Internet services. Note that although worm Slammer attacks Microsoft SQL servers instead of web servers, it also exploits buffer overflow vulnerabilities.

Threshold of Push-calls for Scheme 1 Figure 3.8(a) shows that all instruction sequences distilled from a normal request contain at most one push-call code pattern. Figure 3.8(b) shows that for all the 53 buffer overflow attacks we tested, every attack request contains more than two push-calls in one of its instruction sequences. Therefore, by setting the threshold number of push-calls to 2, Scheme

1 can detect all the attacks used in our experiment.

Threshold of Useful Instructions for Scheme 2 Figure 3.8(c) shows that no normal requests contain an instruction sequence that has more than 14 useful instructions. Figure 3.8(d) shows that an attack request contains over 18 useful instructions in one of its instruction sequences. Therefore, by setting the threshold to a number between 15 and 17, Scheme 2 can detect all the attacks used in our test. The three attacks, which have the largest numbers of instructions (92, 407 and 517), are worm Slammer, CodeRed.a and CodeRed.c, respectively. This motivates us to investigate in our future work whether an exceptional large number of useful instructions indicates the occurrence of a worm.

Threshold of Dependence Degree for Scheme 3 Figure 3.8(e) shows that no normal requests contain an instruction sequence whose dependence degree is more than 3. Figure 3.8(f) shows that for each attack there exists an instruction sequence whose dependence degree is more than 4. Therefore, by setting the threshold to 4 or 5, Scheme 3 can detect all the attacks used in our experiment.

3.5.2 Detection of Polymorphic Shellcode

We also tested polymorphic attack messages from 5 publicly available polymorphic engines: Countdown [10], JumpCallAdditive [10], JempiScodes [8], CLET v1.0 [34] and ADMmutate v0.84 [59]. Polymorphic engines encrypt original exploit code and decrypt them during execution. Countdown uses a decrementing byte as the key for decryption; JumpCallAdditive is a xor decoder; JempiScodes contains three different decryption methods. CLET and ADMmutate are advanced polymorphic engines, which also obfuscate the decryption routine by metamorphism such as instruction replacement and garbage insertion. CLET also uses spectrum analysis to defeat data mining methods.

Because there is no push-call pattern in the code, Scheme 1 cannot detect this type of attacks. However, Scheme 2 and Scheme 3 are still very robust to these obfuscation techniques. This is because although the original shellcode contains more useful instructions than the decryption routine does and it is also encrypted, Scheme 2 may still find enough number of useful instructions and Scheme 3 may still find enough maximum dependence degree in the decryption routines.

We generated 100 different attack messages per each of Countdown, JumpCallAdditive, ADMmutate and CLET. For JempiScodes, we generated 300 different attack messages, 100 per each of its three decryption approaches.

Detection by Scheme 2 We used Scheme 2 to detect the useful instructions in the above 700 attack messages. Figure 3.9(a) shows the (sorted) numbers of useful instructions in 200 polymorphic shellcodes from ADMmutate and CLET. We observed that the least number of useful instructions in these ADMmutate polymorphic shellcodes is 17, whereas the maximum number is 39; the least number of useful instructions in the CLET polymorphic shellcodes is 18, whereas the maximum number is 25. Therefore, using the same threshold value as before (i.e., between 15 and 17), we can detect all the 200 polymorphic shellcodes generated by ADMmutate and CLET. However, for each of the 100 Countdown polymorphic shellcode attack instances, Scheme 2 only reports 7 useful instructions, and for each of the rest polymorphic attack instances, it outputs 11 useful instructions Therefore, if we still use the same threshold (i.e., between 15 and 17), we will not be able to detect these attacks using Scheme 2.

Detection by Scheme 3 We also used Scheme 3 to calculate the dependence degree in the above 700 attack messages. Figure 3.9(b) shows the (sorted) maximum dependence degree in 200 polymorphic shellcodes from ADMmutate and CLET. We observed that the least number of dependence degrees in these ADMmutate polymorphic shellcodes is 7, whereas the maximum number is 22; the least number of dependence degree in the CLET polymorphic shellcodes is 8, whereas the maximum number is 11. The dependence degree of 100 Countdown polymorphic shellcodes is 6. the dependence degree of 100 JumpCallAdditive polymorphic shellcodes is 7; the dependence degree of 100 JempiScodes polymorphic shellcodes is 5. Therefore, by using the same threshold value as before (i.e., 4 or 5), we can detect all the 700 polymorphic shellcodes.

3.5.3 Testing on Real Traces

We next apply SigFree over real traces for false positives.

HTTP Requests in the Local Network Due to privacy concerns, we were unable to deploy SigFree in a public web server to examine real-time web requests.

To make our test as realistic as possible, we deployed a client-side proxy underneath a web browser. The proxy recorded a normal user's http requests during his/her daily Internet surfing. During a one-week period, more than ten of our lab members installed the proxy and helped collect totally 18,569 HTTP requests. The requests include manually typed urls, clicks through various web sites, searchings from search engines such as Google and Yahoo, secure logins to email servers and bank servers, and HTTPs requests. In this way, we believe our data set is diverse enough, not worse than that we might have got if we install SigFree in a single web server that provides only limited Internet services.

Our test based on the above real traces did not yield an alarm. This output is of no surprise because our normal web requests do not contain code.

HTTP Replies in the Local Network To test SigFree over more diverse data types, we also collected totally 17,904 HTTP replies during the above one-week period. The data types of the replies include plain-text, octet-stream, pdf, javascript, xml, shockwave, jpeg, gif, png, x-icon, audio and video. We use each of the three schemes to test over the above replies. Scheme 1 raised warnings on 5 HTTP replies; Scheme 2 raised warnings on 4 HTTP replies; Scheme 3 raised warnings on 10 HTTP replies. By manually checking these warnings, we find they are indeed false positives. The result shows that the three schemes have a few false positives. It also shows although Scheme 3 is more robust to obfuscation than Scheme 1 and 2, it has relatively higher false positives.

CiteSeer Requests Finally we tested SigFree over one-month (September, 2005) 397,895 web requests collected by the scientific and academic search engine Citeseer [3]. Our test based on the Citeseer requests did not yield an alarm.

3.5.4 Performance Evaluation

Stand-alone SigFree We implemented a stand-alone SigFree prototype using the C programming language in the Win32 environment. The stand-alone prototype was compiled with Borland C++ version 5.5.1 at optimization level O2. The experiments were performed in a Windows 2003 server with Intel Pentium 4, 3.2GHz CPU and 1G MB memory. We measured the processing time of the stand-alone prototype over all (2,910 totally) 0-10KB images collected from the above real

traces. We set the upper limit to 10KB because the size of a normal web request is rarely over that if it accepts binary inputs. The types of the images include jpeg,gif,png and x-icon. Figure 3.10 shows that the average processing time of the three schemes increases linearly when the sizes of the image files increase. It also shows that Scheme 1 is the fastest among the three schemes and Scheme 3 is a little bit slower than Scheme 2. In all three schemes, the processing time over a binary file of 10kb is no more than 85ms.

Proxy-based SigFree To evaluate the performance impact of SigFree to web servers, we also implemented a proxy-based SigFree prototype. Figure 3.11 depicts the implementation architecture. It is comprised of the following modules:

URI decoder. The specification for URLs[21] limits the allowed characters in a Request-URI to only a subset of the ASCII character set. This means that the query parameters of a request-URI beyond this subset should be encoded[21]. Because a malicious payload may be embedded in the request-URI as a request parameter, the first step of SigFree is to decode the request-URI.

ASCII Filter. Malicious executable code are normally binary strings. In order to guarantee the throughput and response time of the protected web system, if a request is printable ASCII ranging from 20-7E in hex, SigFree allows the request to pass. Note that ASCII filter does not prevent the service from receiving non-ASCII strings. All non-ASCII strings will analyzed by ISD and ISA. In Section 3.6.2, we will discuss a special type of executable codes called alphanumeric shellcodes [72] that actually use printable ASCII.

The proxy-based prototype was also compiled with Borland C++ version 5.5.1 at optimization level O2. The proxy-based prototype implementation was hosted in the Windows 2003 server with Intel Pentium 4, 3.2GHz CPU and 1G MB memory.

The proxy-based SigFree prototype accepts and analyzes all incoming requests from clients. The client testing traffics were generated by Jef Poskanzer's http_load program [70] from a Linux desktop PC with Intel Pentium 4 2.5GHz CPU connected to the Windows server via a 100 Mbps LAN switch. We modified the original http_load program so that clients can send code-injected data requests.

For the requests which SigFree identifies as normal, SigFree forwards them to the web server, Apache HTTP Server 2.0.54, hosted in a Linux server with dual Intel Xeon 1.8G CPUs. Clients send requests from a pre-defined URL list. The

documents referred in the URL list are stored in the web server. In addition, the prototype implementation uses a time-to-live based cache to reduce redundant HTTP connections and data transfers.

Rather than testing the absolute performance overhead of SigFree, we consider it more meaningful measuring the impact of SigFree on the normal web services. Hence, we measured the *average response latency* (which is also an indication of *throughput* although we did not directly measure throughput) of the connections by running http_load for 1000 fetches. Figure 3.12(a) shows that when there are no buffer overflow attacks, the average response time in the system with SigFree is only slightly higher than the system without SigFree. This indicates that, despite the connection and ASCII checking overheads, the proxy-based implementation does not affect the overall latency significantly.

Figure 3.12(b) shows the average latency of connections as a function of the percentage of attacking traffic. We used CodeRed as the attacking data. Only successful connections were used to calculate the average latency; that is, the latencies of attacking connections were not counted. This is because what we care is the impact of attack requests on normal requests. We observe that the average latency increases slightly worse than linear when the percentage of malicious attacks increases. Generally, Scheme 1 is about 20% faster than Scheme 2 and Scheme 3 is slightly slower than Scheme 2.

Overall, our experimental results from the prototype implementation show that SigFree has reasonably low performance overhead to web servers. Especially when the fraction of attack messages is small (say $< 10\%$), the additional latency caused by SigFree to web servers is almost negligible.

3.6 Discussion

3.6.1 Robustness to Obfuscation

Most malware detection schemes include two-stage analysis. The first stage is disassembling binary code and the second stage is analyzing the disassembly results. There are obfuscation techniques to attack each stage[57, 30] and attackers may use them to evade detection. Table 3.1 shows that SigFree is robust to most of

these obfuscation techniques.

Obfuscation in The First Stage *Junk byte insertion* is one of the simplest obfuscation against disassembly. Here junk bytes are inserted at locations that are not reachable at run-time. This insertion however can mislead a linear sweep algorithm, but cannot mislead a recursive traversal algorithm[52], on which our algorithm bases.

Opaque predicates are used to transform unconditional jumps into conditional branches. Opaque predicates are predicates that are always evaluated to either true or false regardless of the inputs. This allows an obfuscator to insert junk bytes either at the jump target or in the place of the fall-through instruction. We note that opaque predicates may make SigFree mistakenly interpret junk byte as executable codes. However, this mistake will not cause SigFree to miss any real malicious instructions. Therefore, SigFree is also immune to obfuscation based on opaque predicates.

Obfuscation in The Second Stage Second-stage obfuscation techniques can easily obfuscate the rule-based scheme by Chinchani and Berg [24]. For example, Rule 1 expects push instructions appears before the branch. This Rule can be confused by obfuscation technique instruction replacement (e.g. "push eax" replaced by "sub esp 4 ; move [esp] eax"). Most of the second-stage obfuscation techniques obfuscate the behaviors of a program; however, the obfuscated programs still bear characteristics of programs. Since the purpose of SigFree is to differentiate executable codes and random binaries rather than benign and malicious executable codes, most of these obfuscation techniques are ineffective to SigFree. Obfuscation techniques such as instruction reordering, register renaming, garbage insertion and reordered memory accesses do not affect the number of calls or useful instructions which our schemes are based on. By exploiting instruction replacement and equivalent functionality, attacks may evade the detection of Scheme 1, but cannot evade the detection of Scheme 2 and Scheme 3.

3.6.2 Limitations

SigFree also has several limitations. First, SigFree cannot fully handle the branch-function–based obfuscation, as indicated in Table 3.1. Branch function is a function

Table 3.1. SigFree is robust to most obfuscation

Disassembly stage	Obfuscation	SigFree	
	Junk byte insertion	Yes	
	Opaque predict	Yes	
	Branch function	partial	
Analysis stage	Obfuscation	Scheme 1	Schem 2&3
	Instruction reordering	Yes	Yes
	Register renaming	Yes	Yes
	Garbage insertion	Yes	Yes
	Instruction replacement	No	Yes
	Equivalent funcationality	No	Yes
	Reordered memory accesses	Yes	Yes

$f(x)$ that, whenever called from x, causes control to be transferred to the corresponding location $f(x)$. By replacing unconditional branches in a program with calls to the branch function, attackers can obscure the flow of control in the program. We note that there are no general solutions for handling branch function at the present state of the art.

With respect to SigFree, due to the obscurity of the flow of control, branch function may cause SigFree to break the executable codes into multiple instruction sequences. Nevertheless, it is still possible for SigFree to find this type of buffer overflow attacks as long as SigFree can still find enough number of useful instructions or dependence degree in one of the distilled instruction sequences.

Second, SigFree cannot fully handle self-modifying code. Self-modifying code is a piece of code which dynamically modifies itself at runtime and could make SigFree mistakenly exclude all its instruction sequences. It is crafted in a way such that static analysis will reach illegal instructions in all its instruction sequences, but these instructions become legal during execution. To address this attack, we may remove Step 3 in excluding instruction sequences; that is, we do not use inevitably reachable illegal instructions as a criterion for pruning instruction sequences. This however will increase the computational overhead as more instruction sequences will be analyzed. Self-modifying code can also reduce the number of *useful* in-

structions, because some instructions are considered *useless*. We note that there are also no general static solutions for handling self-modifying code at the present state of the art. Nevertheless, it is still possible for SigFree to detect self-modifying code, because self-modifying code itself is a piece of code which may have enough number of useful instructions or dependence degree. Our future work will explore this area.

Third, the executable shellcodes could be written in alphanumeric form[72]. Such shellcodes will be treated as printable ASCII data and thus bypass our analyzer. By turning off the ASCII filter, Scheme 2 and Scheme 3 can successfully detect alphanumeric shellcodes; however, it will increase computational overhead. It therefore requires a slight tradeoff between tight security and system performance.

Fourth, SigFree does not detect attacks such as return-to-libc attacks that just corrupt control-flow or data without injecting code. However, these attacks can be handled by some simply methods. For example, return-to-libc attacks can be defeated by mapping (through mmap()) the addresses of shared libraries so that the addresses contain null bytes[2][33].

Finally, it is still possible that attackers evade the detection of Scheme 3 by using specially-crafted code, once they know the threshold of Scheme 3. However, we believe it is fairly hard, as the bar has been raised. For example, it is difficult, if not impossible, to reduce the dependence degree to 3 or fewer in all the instructions of Countdown decryption routine showed in Figure 3.7(a). Decryption instruction a depends on all other instructions in Figure 3.7(b). These instruction can be classified into three groups: instructions 0,2 and 7 are used to initialize loop counter register ecx; instruction e is used as a loop instruction; instructions 3 and 9 are used to get PC (program counter). To reduce the dependency degree of instruction a to 3, attackers have to use one instruction to initialize the loop counter and one instruction to get PC. However, attackers cannot use one instruction to initialize loop counter (e.g. mov ecx, 0x0000005B), which contains null bytes. It is also hard to find a single instruction to get PC because IA-32 architecture does not provide any instruction to directly read PC.

[2]Null bytes, which are C string terminators, cause attacks terminate before they overflow the entire buffer

3.6.3 Application-Specific Encryption Handling

The proxy-based SigFree could not handle encrypted or encoded data directly. A particular example is SSL-enabled web server. Enhancing security between web clients and web servers by encrypting HTTP messages, SSL also causes the difficulty for out-of-box malicious code detectors.

To support SSL functionality, an SSL proxy such as Stunnel[13] (Figure 3.13) may be deployed to securely tunnel the traffic between clients and web servers. In this case, we may simply install SigFree in the machine where the SSL proxy is located. It handles the web requests in cleartext that have been decrypted by the SSL proxy. On the other hand, in some web server applications, SSL is implemented as a server module (e.g., mod_ssl in Apache). In this case, SigFree will need to be implemented as a server module (though not shown in Figure 3.13), located between the SSL module and the WWW server. We notice that most popular web servers allow us to write a server module to process requests and specify the order of server modules. Detailed study will be reported in our future work.

3.6.4 Applicability

So far we only discussed using SigFree to protect web servers. It is worth mentioning that our tool is also widely applicable to many programs that are vulnerable to buffer overflow attacks. For example, the proxy-based SigFree can be used to protect all Internet services which do not permit executable binaries to be carried in requests. SigFree should not directly be used to protect some Internet services that do accept binary code such as FTP servers; otherwise SigFree will generate many false positives. To apply SigFree for protecting these Internet services, other mechanisms such as whitelisting need to be used.

In addition to protecting servers, SigFree can also provide file system real-time protection. Buffer overflow vulnerabilities have been found in some famous applications such as Adobe Acrobat and Adobe Reader[12], Microsoft JPEG Processing (GDI+)[2], and WinAmp[16]. This means that attackers may embed their malicious code in PDF, JPEG, or mp3-list files to launch buffer overflow attacks. In fact, a virus called Hesive[14] was disguised as a Microsoft Access file to exploit

buffer overflow vulnerability of Microsoft's Jet Database Engine. Once opened in Access, infected .mdb files take advantage of the buffer overflow vulnerability to seize control of vulnerable machines. If mass-mailing worms exploit these kinds of vulnerabilities, they will become more fraudulent than before, because they may appear as pure data-file attachments. SigFree can be used alleviate these problems by checking those files and email attachments which should not include any code.If the buffer being overflowed is inside a JPEG or GIF system, ASN.1 or base64 encoder, SigFree cannot be directly applied. Although SigFree can decode the protected file according to the protocols or applications it protects, more details need to be studied in the future.

Although SigFree is implemented in the Intel IA32 architecture, it is a generic technique, which can be ported to other platforms such as PowerPC. The mechanism of code abstraction technique and its robustness to obfuscation are not related to any hardware platform. Therefore, we believe that detection capabilities and resilience to obfuscation will be preserved after porting. Of course, some implementation details such as handling special cases in Scheme 2 need to be changed. We will study this portability issue in our future work.

Finally, as a generic technique SigFree can also block other types of attacks as long as the attacks perform binary code injection. For example, it can block code-injection format string attacks.

3.7 Conclusion

We proposed SigFree, a on-line, signature free, out-of-the-box blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats. SigFree does not require any signatures, thus it can block new, unknown attacks. SigFree is immunized from most attack-side code obfuscation methods, good for economical Internet wide deployment with little maintenance cost and low performance overhead.

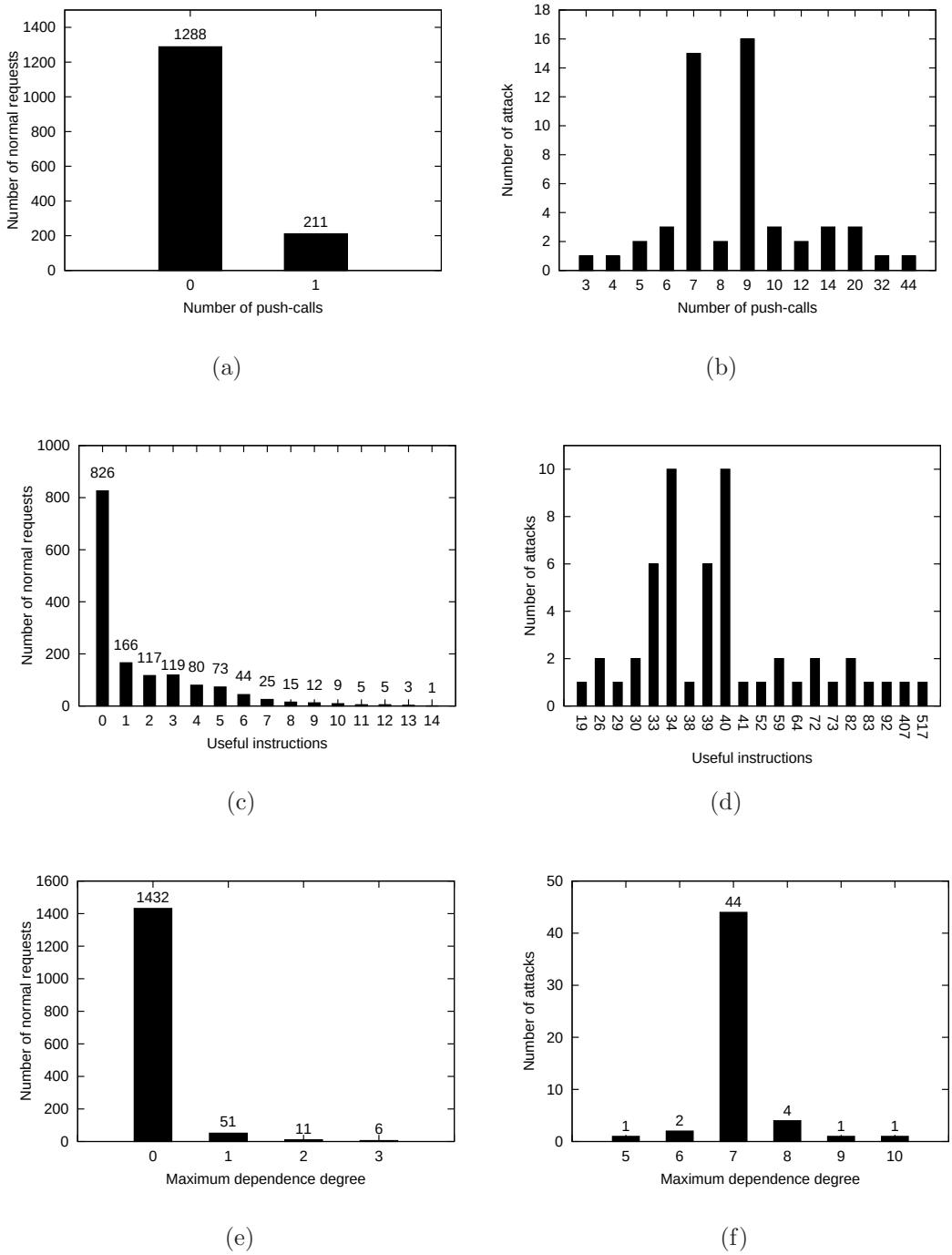
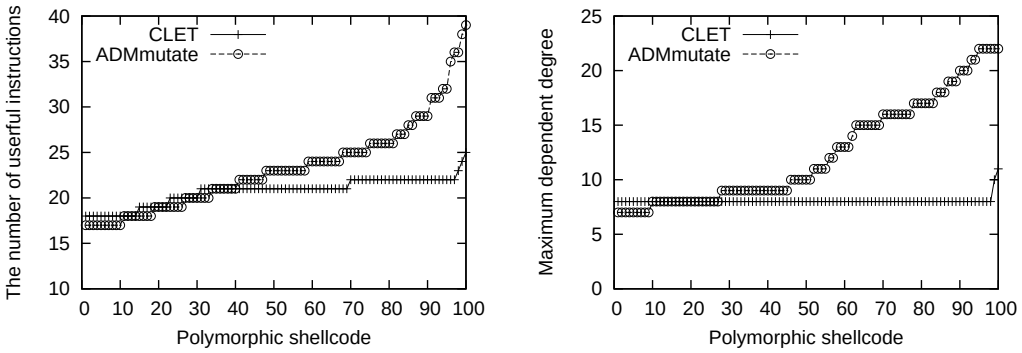


Figure 3.8. (a) The number of push-calls in normal requests (b) The number of push-calls in attack requests (c) The number of useful instructions in normal requests (d) The number of useful instructions in attack requests (e) Maximum dependence degree in normal requests (f) Maximum dependence degree in attack requests in a request.



(a) The number of useful instructions (b) Maximum dependence degree

Figure 3.9. The number of useful instructions and maximum dependence degree in all 200 polymorphic shellcodes.

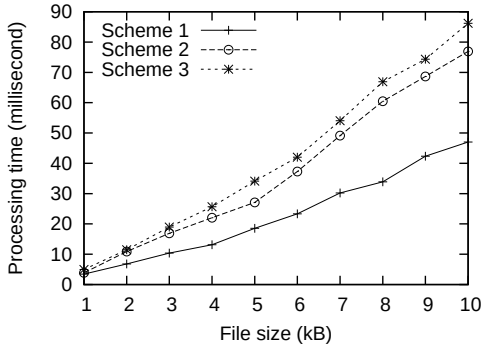


Figure 3.10. Processing time of SigFree over image files of various sizes

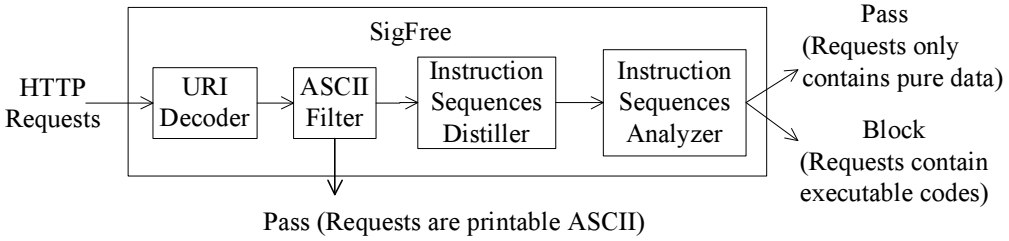
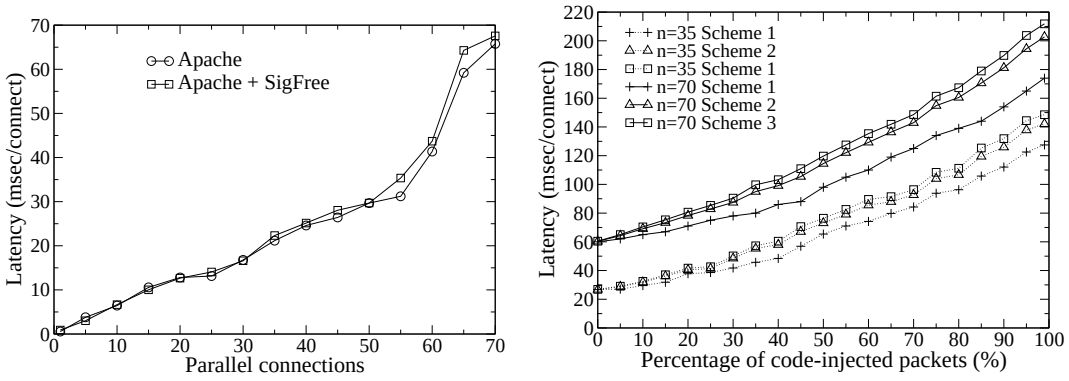


Figure 3.11. The architecture of SigFree



(a) (b)

Figure 3.12. Performance impact of SigFree on Apache HTTP Server

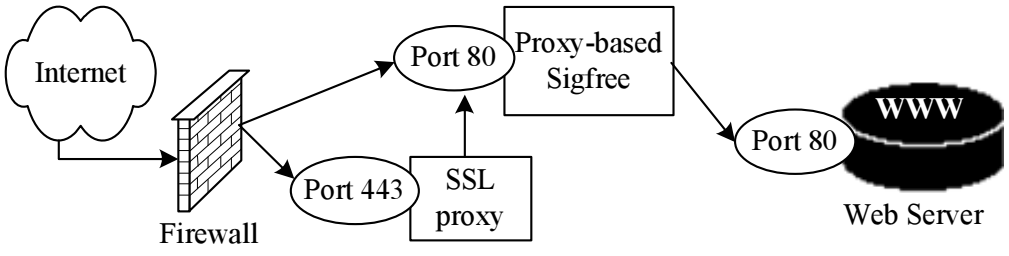


Figure 3.13. SigFree with a SSL proxy

Chapter 4

STILL: Exploit Code Detection via Static Taint and Initialization Analyses

4.1 Introduction

Besides triggering human beings to do such unwise things as opening malicious email attachments, disclosing their passwords and identities, and visiting fake web sites, human-intervention-free remote exploit code is a primary vehicle for attackers to harvest bots and launch various attacks. The most serious vulnerability exploited by binary exploit code is buffer overflow, which exists pervasively in Internet services (e.g., port 80 Web service), OS services (e.g., Windows DCE-RPC or CIFS/SMB), database services, applications (e.g., browser plug-ins), and so on. Buffer overflow vulnerabilities allow attackers to use a network request to inject a piece of exploit code into the "body" of a service or application program. Once such exploit code is executed, the attacker may gain full control of the victim machine. In different attacks, exploit code may be in different forms: a piece of shellcode to break into a certain type of hosts, an infection vector for Internet worms such as CodeRed and Slammer, and so on. From both CERT [4] and Microsoft Security Bulletin [11] we can clearly see that the majority of vulnerabilities/attacks in the Microsoft Windows family are buffer-overflow based automatic remote code

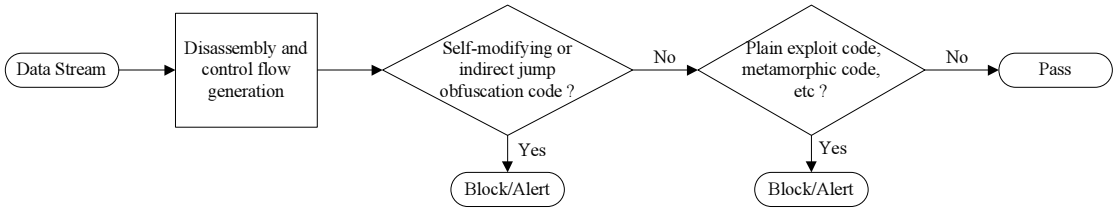


Figure 4.1. The architecture of STILL

execution.

Network intrusion detection systems (NIDSes) such as Snort [15] use manually prepared string-matching signatures to detect code-injection attacks. To contain zero-day attacks, automatic signature generation techniques have recently been proposed to automatically extract string-matching signatures from malicious payloads. Some representative examples of these techniques are EarlyBird [76], Polygraph [65], Hamsa [54], and Packet Vaccine [88]. However, a common limitation of string-matching signature based defenses, whether their signatures are manually prepared or automatically generated, is that they (e.g., [85, 76]) or their flow classifiers (e.g., [54]) are not very resilient to code-obfuscation.

To address the limitations of the signature-based defense mentioned above, researchers have recently proposed several static analysis approaches to identify and analyze the code contained in buffer overflow attack packets [51, 24]. These approaches are based on the observation that remote exploits typically contain executables, whereas legitimate client requests never contain executables in most Internet services such as Web services and SQL services. These static analysis approaches can be divided into two stages. The first stage is to distill instruction sequences from network packets by disassembly. These instruction sequences may be real code or random instructions. The second stage is to analyze the disassembly results by exploiting control flow [51] or data flow analysis [24] to discern code/malicious code from data. Compared with string-matching signature based approaches, static analysis approaches have two notable advantages. First, they can detect new (or zero-day) exploit code for both known and unknown vulnerabilities. Second, they are more resilient to polymorphism and metamorphism.

However, by applying some anti-static-analysis obfuscation techniques such as self-modifying and indirect jump[69, 57], an attacker can evade these static analysis

approaches such as SigFree [87] and [51, 24]. There are two main techniques to thwart static analysis [69]. One aims at thwarting disassembly via indirect jump. An indirect jump instruction transfers control to the address contained in a register operand and its destination is hard to be statically determined. The other technique is to thwart control flow and data flow analyses by self-modifying code. Control transfer instructions (CTIs) can be dynamically changed to non-CTIs at runtime by self-modifying code, thus obfuscating control flows. Data transfer/operation instruction can be changed to other instructions dynamically at runtime by self-modifying code, thus obfuscating data flows.

Our Approach We propose *STILL, a novel static taint and initialization analysis based approach, to detect not only unobfuscated exploit code (without any obfuscation), traditional polymorphic and metamorphic exploit code, but also self-modifying and indirect jump obfuscation code.* STILL is based on the same observation as in [51, 24, 87] that remote exploits typically contain executables, whereas legitimate client requests never contain executables in most Internet services.

STILL detects attacks as follows. It works as a proxy-based blocker in the application layer of clients and/or servers. When it captures a data stream, it disassembles the data stream and generates a control flow graph. It analyzes the disassembled result in two stages. First, STILL detects self-modifying and indirect jump obfuscation code. Although the real exploit code may be hidden by self-modifying and indirect jump, the obfuscation code itself provides some strong evidence of self-modifying and/or indirect jump behavior. STILL detects this behavior by static taint analysis and initialization analysis. Since polymorphism is a kind of self-modifying, STILL can also detect polymorphic code in this stage. Of course, an attacker might use neither self-modifying nor indirect jump obfuscation. Hence, in the second stage, STILL tries to detect the plain exploit code, which may even have been obfuscated by metamorphism. STILL also exploits static analysis and initialization analysis in this stage to combat other obfuscation techniques. Figure 4.1 depicts the architecture of STILL.

Contributions The main merits of STILL are as follows. First, STILL (including *static taint analysis* and *initialization analysis*) is a static analysis technology that can detect both self-modifying code and indirect jump, whereas pre-

vious static analysis approaches [51, 24, 87] could be easily thwarted by these obfuscation techniques. Second, STILL is robust to almost all anti-signature, anti-static-analysis and anti-emulation obfuscation. Third, STILL is a more comprehensive solution for detecting zero-day exploit code, regardless of whether the exploit code is obfuscated or not. Note that although the proposed techniques, like many intrusion detection systems, somehow look heuristic, we believe they are very fundamental and greatly raise the bar for future attacks.

A prototype implementation of STILL has been developed. Experimental results show that STILL successfully detected all 12,000 polymorphic shellcode payloads generated by 10 well-known polymorphic engines, all 34 plain Linux and Windows shellcodes generated by the Metasploit framework, and 5 worms (CodeRed I, CodeRed II, Sasser, Blaster, and Slammer) that are available to us. Moreover, our large-scale testing shows that STILL has an extremely low false positive rate.

Organization The rest of the chapter is organized as follows. In Section 4.2, we give an overview of exploit code obfuscation. In Section 4.3, we show how we disassemble a data stream and then generate a control flow graph (4.3.1), the approach to detecting self-modifying and indirect jump obfuscation code (4.3.2), and the approach to detecting plain exploit code (4.3.3). We show the strength of STILL and discuss the limitations in Section 4.4. In Section 4.5, we show our experimental results. Finally, we conclude the chapter in Section 4.6.

4.2 Obfuscation of Exploit Code

In this section, we give a brief overview of the known techniques for obfuscating exploit code. We classify these obfuscation techniques into three categories according to their purposes: anti-signature, anti-static-analysis and anti-emulation. We show how anti-static-analysis thwarts previous static analysis techniques such as SigFree [87] and others [51, 24]. We will discuss the effectiveness of STILL in handling these obfuscation techniques in Section 4.4.

4.2.1 Anti-signature

Anti-signature obfuscation is the type of obfuscation techniques for thwarting traditional string-matching signature based detection. It includes metamorphism and

polymorphism. Metamorphism is the type of obfuscation techniques which evade signature-based detection by instruction reordering, register renaming, garbage insertion, instruction replacement and equivalent functionality. More details of these techniques can be found in [30, 61].

Polymorphism is a type of obfuscation techniques which normally rely on encryption to mutate code. Original code is encrypted and hidden in the payload, and it is decrypted to its original form during execution. However, a small portion of it, the decryption routine is left unencrypted. Some polymorphic engines further rewrite the unencrypted decryption routine by metamorphism each time the exploit code is propagated [34, 59].

4.2.2 Anti-static-analysis

Anti-static-analysis is the type of obfuscation techniques for thwarting static analysis. It can be divided into three subcategories: anti-disassembly, self-modifying and memory access obfuscation. Note that some anti-static-analysis techniques such as self-modifying and indirect jump can easily thwart previous static analysis techniques [51, 24, 87]. Next, we describe these techniques in slightly more details.

Self-Modifying Self-modifying code is code that modifies itself when being executed. Self-modifying is a very powerful technique to thwart static analysis since it can completely hide the semantics of original instructions. More specifically, it may be used to thwart static analysis in a number of ways. First, it can be used to obfuscate the control flow graph (CFG) of exploit code, while CFG is the obligatory input to a static analysis approach. As an example, in Figure 4.2(a) the original loop instruction at address 16 is replaced in Figure 4.2(b) by a nop instruction at $1d$ and an invalid instruction at $1e$, and these two instructions will be modified back to the loop instruction at runtime by instruction $0c$ in Figure 4.2(b). Second, it can be used to obfuscate data flow of exploit code to thwart those data flow analysis based approaches such as SigFree [87] and [24]. For example, a data transfer instruction can be changed to a nop instruction dynamically at runtime by self-modifying code, thus obfuscating data flows. Note that polymorphic code is also a kind of self-modifying code, which hides original exploit code. For example, in Figure 4.2(a) the encrypted payload (encrypted original exploit code) will be decrypted at runtime and then be executed.

```
00:  2BC9                sub ecx,ecx
02:  83E9 B0             sub ecx,-0x50
05:  E8 FFFFFFFF         call 0x09
09:  FFC0                inc eax
0b:  5E                  pop esi
0c:  8176 0E DC40D776    xor [esi+0xE],0x76D740DC
13:  83EE FC             sub esi,-0x4
16:  E2 F4               loopd 0x0C
18:  (Encrypted payload)
     ...
```

(a)

```
00:  2BC9                sub ecx,ecx
02:  83E9 B0             sub ecx,-0x50
05:  E8 FFFFFFFF         call 0x09
09:  FFC0                inc eax
0b:  5E                  pop esi
0c:  668176 0A 7264      xor [esi+0x13],0x0A72
12:  8176 0E DC40D776    xor [esi+0x15],0x76D740DC
19:  83EE FC             sub esi,-0x4
1d:  90                  nop
1e:  FE                  (invalid instruction)
1f:  (Encrypted payload)
     ...
```

(b)

Figure 4.2. (a) A decryption routine generated by Engine Pex [10]. (b) The decryption routine obfuscated by self-modifying which confuses control flows.

Anti-disassembly is the type of obfuscation techniques that try to confuse traditional disassembly algorithms (e.g., linear sweep [57]). It includes junk byte insertion, opaque predicate, code overlap, indirect jump and branch function.

Indirect jump transfers control to the instruction whose address is in a register operand. Because the value of the register may not be statically determined, disassembly algorithm such as recursive traversal cannot provide an accurate disassembly. Therefore, attackers may use indirect jump to replace relative jump in the payload. For example, relative jump instruction "jmp 0x05" can be replaced

by indirect jump instruction "jmp eax", where eax contains absolute address of the target. The value of eax is normally hard to determine until at run-time, thus thwarting static analysis.

Branch function is a function $f(x)$ that, whenever called from x, causes control to be transferred to the corresponding location $f(x)$ [57]. By replacing unconditional branches in a program with calls to a branch function, attackers can obscure the flow of control in the program.

Memory Access Obfuscation is the type of obfuscation techniques that use indirect addressing for memory access to thwart static analysis. For example, instruction "mov ebx,[ss:esp]", which moves the top item of stack into ebx, may be obfuscated by instructions " mov eax,esp; mov ebx,[ss:eax]".

4.2.3 Anti-emulation

There are many anti-emulation techniques, such as using interrupts in polymorphic decryption routines, inserting delay loops, executing random code. These techniques have existed in virus writer community for many years and more details can be found in [79].

4.3 A Generic Code Detection Technique

In this section, we present STILL in three steps, as shown in Fig. 4.1.

4.3.1 Disassembly and Control Flow Graph Generation

The first step of STILL is to disassemble the input data stream and generate a control flow graph. A major challenge here is that we do not know whether the data stream contains code or not, and what the entry point of the code is when code is present. As such, it is not directly clear which parts of a stream should be disassembled. The problem is exacerbated by the fact that different types of instructions have varying lengths and most bit combinations map to valid instructions in the Intel IA32 instruction set. In fact, even a stream of random bytes (or a part of a stream) could be disassembled into a valid instruction sequence[51, 87].

In previous work[51, 24, 87] several disassembly algorithms have been proposed to address the aforementioned challenges. In STILL, we exploit the O(N) disassembly algorithm used in [87], where N is the length of the data stream. This algorithm will result in a set of instruction sequences. An *instruction sequence* is a sequence of CPU instructions which has one and only one entry instruction and there exists at least one execution path from the entry instruction to any other instruction in this sequence. A fragment of a program in machine language is an instruction sequence, but an instruction sequence is not necessarily a fragment of a program. In fact, we may distill (random) instruction sequences from any binary strings (e.g., a GIF file). Two example instruction sequences are shown in Figure 2: sequence 1 includes instructions 00 to 16 in Figure 4.2(a); sequence 2 includes instructions 00 to 1d in Figure 4.2(b).

STILL is more robust to anti-disassembly techniques than previous work[51, 24, 87]. For example, previous work exploits some heuristics to prune basic blocks. In [51], a basic block is considered invalid in three cases: (1) if it contains one or more invalid instructions; (2) if it is on a path to an invalid block; (3) if it ends in a control transfer instruction that jumps into the middle of another instruction. In [87], some similar heuristics are applied. STILL does not use the first two heuristics because invalid instructions could be the result of self-modifying obfuscation. For example, the basic block (instructions from 00 to 1e in Figure 2), which ends with an invalid instruction, should not be pruned. STILL does not use the third heuristic either because a control transfer instruction may jump into the middle of another instruction by using code overlap obfuscation [69].

We note that in the presence of indirect jump and self-modifying obfuscation, it is impossible to completely and statically disassemble the entire body of the exploit code embedded in a data stream using the recursive traversal algorithm. Fortunately, the partially disassembled result may already provide some strong evidence of self-modifying and/or indirect jump behavior. In Figure 4.2(b), neither the decryption routine (e.g., the loop instruction can no longer be seen in the disassembly result) nor the original exploit shellcode can be successfully disassembled. However, the instructions from 00 to 1d indicate the self-modifying behavior. Our approach to detecting self-modifying and indirect jump code in the next section will only use this partially disassembled result as the input.

4.3.2 Detection of Self-modifying and Indirect Jump Obfuscation Code

Once the (partial) instruction sequences have been extracted in the first step (i.e., the disassembly process), the next step is to determine whether they are real exploit code. As we showed in Section 4.2, there are many techniques for obfuscating shellcode. In this section, however, our discussion will concentrate on detecting self-modifying and indirect jump exploit code, because these two types of exploit code can evade the detection of previous static analysis schemes [87, 24, 51] and are very challenging to detect. Clearly, if STILL can detect these two types of exploit code, it will also be capable of detecting branch function and polymorphic code because branch function uses an indirect jump to transfer control to the original target and polymorphic code is a kind of self-modifying code. In Section 4.4 we will show that STILL is rather effective in handling the other types of obfuscation.

The new techniques in STILL to detecting self-modifying and indirect jump exploit code are called *static taint analysis* and *initialization analysis*. We observe that self-modifying and indirect jump exploit codes first need acquire the absolute address of payload. Then, the absolute address will be used in a certain way (referred to as *abstract semantics*) reflecting self-modifying and indirect jump behavior, whereas this behavior is very rare in random instruction sequences. Accordingly, self-modifying and indirect jump exploit codes are detected as follows. First, the variable which holds the absolute address of the payload is found in the instruction sequences and used as a *taint seed*. Then, static taint analysis is used to track the tainted values and detect whether tainted data are used in the abstract semantics that could indicate the presence of self-modifying and indirect jump exploit code. Finally, we use initialization analysis to reduce false positives.

Taint Seed The absolute address of payload is used as a taint seed in STILL. Both indirect jump and self-modifying obfuscation code need the absolute address of payload, because the target for indirect jump and memory read/write for self-modifying must use absolute addresses in IA-32 architecture [7]. They have to know their own absolute address in order to jump within the payload or modify the payload at runtime. There are two reasons that attackers want to get the absolute address of payload at runtime rather than predicting it or hardcoding it.

First, the address of payload cannot be predicted in most cases because exploit code is placed in a dynamically changing stack or heap [69]. Second, even if attackers can sometimes get the address, it is not good practice to hardcode it, especially in the case of a worm [77]. The absolute address of payload could be different for different versions and different patches of the same operating system (e.g. Windows 2000 with Service Patch 1, Patch 2 and Windows XP Service Patch 1); hence, hardcoding the absolute address could greatly limit the broad spread of a worm.

The only way to get the absolute address of payload is to read the PC (Program Counter) register, which stores the absolute address of the next instruction to be executed. Since the IA-32 architecture does not provide any instruction to directly read PC, attackers have to acquire it at runtime. To the best of our knowledge, currently only three ways (called getPC) are used in the attacker community. First, attackers may use a relative call. Whenever a call is executed, the return address is pushed into the stack just before the control is transferred to the call target. Therefore, the top item of the stack is used as the taint seed at the relative call target. Second, the attackers can also get the address by using the *fstenv* instruction, which saves the current FPU (Floating Point Unit) operating environment at the memory location specified by its operand [67]. The FPU operating environment includes the instruction pointer of the last executed float point instruction. Hence, we can also find the taint seed by checking a float point instruction and a succeeding fstenv instruction. Finally, attackers may also get the address by using the structured exception handling (SEH) mechanism of Windows [42]. However, as mentioned in [69], this technique is feasible only with older versions of Windows. In this chapter, we do not consider this case. Note that whenever a new way for getting PC (if exist) is found, we can easily add a corresponding method in STILL to find the taint seed while the rest parts of STILL will not be affected by this update.

Static Taint Analysis After the taint seed is found, a new *static taint analysis* approach is used to statically determine which variables are tainted in an instruction sequence. A taint seed itself is a tainted variable.

A tainted variable is propagated to a new tainted variable by data transfer instructions that move data (e.g., push, pop, move) and data operation instructions

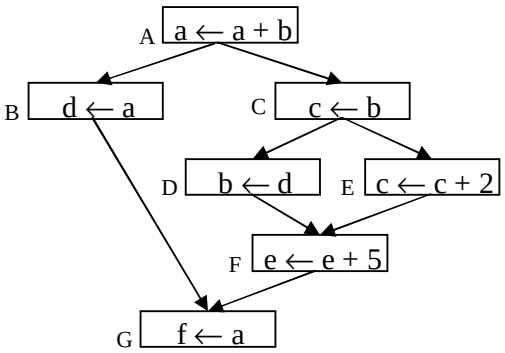
that perform arithmetic or bit-logic operations on data (e.g., add, sub, xor) in the IA-32 instruction set. Other instructions such as control transfer instructions do not affect the taint process. For data transfer instructions, the destination operand will be tainted if and only if the source operand is tainted. For data operation instructions, the destination operand will be tainted if and only if either the source or the destination operand is tainted. Note that for data movement and arithmetic instructions, constants are considered untainted.

If an instruction sequence is straight-line code (i.e., no jump in code), the static taint algorithm is trivial: to taint an instruction sequence we taint the straight-line instructions one by one. However, most instruction sequences (either random or not) contain forward branches and backward branches, which create multiple paths in the code. A variable is tainted if it is tainted in any one of the paths. We define a tainted set equation as following:

$$T(i) = \cup_{x \in pred(i)} (\text{if } (USE(x) \cup T(x) \neq \emptyset) \text{ then } T(x) \cup DEF(x) \text{ else } T(x) - DEF(x)),$$

where i denotes an instruction, $T(i)$ is the set of tainted variables from the entry to instruction i, $Pred(i)$ is the set of i's predecessor instructions in the CFG, $USE(x)$ is the set of variables used in instruction x; $DEF(x)$ is the set of variables defined by instruction x. Figure 4.3 shows an example of calculating T(i) for every instruction.

If an instruction sequence contains backward branches, it is not possible to calculate $T(i)$ for all instructions in one pass because the tainted set of later instructions may affect the tainted set of earlier instructions. This means we have to iteratively pass the instruction sequence several times to recalculate the $T(i)$ until it becomes stable. In STILL, we adopt a classic worklist iterative algorithm [37], where the worklist is the specific iterative scheme. When the algorithm starts, the worklist only contains the successors of the entry node in the instruction sequence. A node denoted by i is removed from the worklist every time and $T(i)$ is recomputed. When the $T(i)$ changes, the algorithm adds to the worklist all of i's successors that are not in the worklist yet. When the worklist is empty, the algorithm terminates. Note that when specific termination conditions are met, e.g., self-modifying is detected, the static taint algorithm may terminate before the worklist becomes empty. The algorithm is sketched in Algorithm 3.



(a)

	A	B	C	D	E	F	G
USE	a,b	a	b	d	c	e	a
DEF	a	d	c	b	c	e	f

(b)

T(A) = {b}

T(B) = T(A) ∪ DEF(A) (USE(A) ∩ T(A) ≠ ∅)
= {a,b}

T(C) = {a,b}

T(D) = T(C) ∪ DEF(C) (USE(C) ∩ T(C) ≠ ∅)
= {a,b,c }

T(E) = {a,b,c }

T(F) = (T(D)– DEF(D)) ∪ (T(E) ∪ DEF(E))
(USE(D) ∩ T(D) = ∅, USE(E) ∩ T(E) ≠ ∅)
= {a,c} ∪{a,b,c} = {a,b,c}

T(G) = (T(B) – DEF(B)) ∪ (T(F) – DEF(F))
(USE(B) ∩ T(B) = ∅, USE(F) ∩ T(F) ≠ ∅)
= {a,b,d} ∪{a,b,c} = {a,b,c,d}

(c)

Figure 4.3. An example of calculating T(i) at each instruction. Assume that only b is tainted at the entry instruction A. (a) The example of a control flow graph; (b) The table of USE and DEF sets of each instruction; (c) The calculated T(i) for each instruction.

Detection by Abstract Semantics Certain abstract semantics can be observed from the tainted data of a real instruction sequence, whereas these abstract semantics are very rare in random instruction sequences. Hence, if these abstract semantics are detected through static taint analysis, an alert will be raised. The following are the abstract semantics for self-modifying and indirect jump, respectively.

Self-modifying Self-modifying obfuscation works in three steps. First, it reads the payload; second, it modifies the read result; finally, it writes the modified result back to the payload. Attackers may implement these three steps in two ways. One way is to use a single updating instruction in the payload to implement these three steps. Instruction 0c in Figure 4.2(b) is such an example. The other way uses several instructions, one instruction for reading payload, several instructions for modifying the read results and one instruction for writing the modified results back to payload. The CLET [34] shellcode generation engine uses this approach. Accordingly, there are two cases where tainted data indicate self-modifying obfuscation. First, the tainted data are used as the address of the updating instructions. Second, the tainted data are used as the address of a memory read instruction or the address of a memory write instruction. We note that the read result will be

Algorithm 3 Static Taint Analysis Algorithm

Input: A instruction sequence $i_0 ... i_N$; A tainted set C in the beginning.
Output: $T(i)$ for each instruction i.
Procedure:

1: $T(i_0) \leftarrow C$
2: **for** $j = 1$ to N **do**
3: $\quad T(i_j) \leftarrow \emptyset$
4: **end for**
5: $worklist \leftarrow$successors(i_0)
6: **while** $worklist \neq \emptyset$ **do**
7: $\quad$ remove an instruction k from $worklist$
8: $\quad$ recompute $T(k)$ as $T(k) \leftarrow \cup_{x \in pred(k)}$ (if $(USE(x) \cup T(x) \neq \emptyset)$ then $T(x) \cup DEF(x)$ else $T(x) - DEF(x)$)
9: $\quad$ **if** $T(k)$ changed **then**
10: $\quad\quad worklist \leftarrow worklist \cup$ successors(k)
11: $\quad$ **end if**
12: **end while**

used to generate the write result; therefore, we start a new taint analysis process to taint the read result. If the newly tainted data are used as the source operand of a memory write instruction, it clearly indicates self-modifying obfuscation. Figure 4.4 shows these two ways of identifying self-modifying code through static taint analysis.

Indirect jump To detect indirect jump obfuscation, we check whether tainted data are used as target addresses of control transfer instructions such as branch, return, and function call instructions. Normally, it is rare that tainted data are used as jump targets in random instruction sequences.

Reducing False Positives Although for random instruction sequences it is not common that the tainted data are used in the same way as the way they are used by self-modifying and indirect jump, we still find some false positives in our experiments. To reduce false positives, we further use *initialization analysis*. We observed that the operands of self-modifying and indirect jump code must be initialized. Specifically, target addresses of indirect jump should be initialized; the operands of memory updating or writing instructions in self-modifying code should be initialized. If these operands are uninitialized, we will not consider them as attacks.

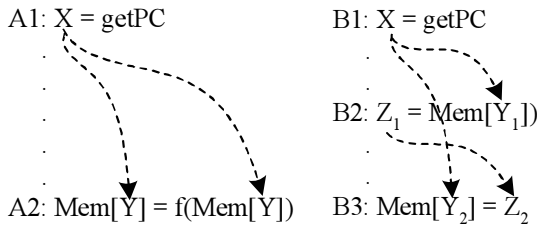


Figure 4.4. (a) The tainted data are used as the addresses of the updating instructions. Variable X is tainted at A_1 at the beginning. Variable Y is tainted by X and used as the address of the updating instruction A_2. (b) Attackers use a memory read instruction to read payload, modify the read result, and write the modified result back to the payload by using a memory write instruction. Variable X is tainted at B_1 at the beginning. Variable Y_1 is tainted by X and used as the address of memory read instruction B_2. Variable Y_2 is tainted by X and used as the address of memory write B_3. Variable Z_2 is tainted by Z_1 and used as the source operand of B_3.

We say a variable is initialized if it is defined by a constant or other initialized variables. It is hard for attackers to know the run-time values of registers before malicious code is executed. That is, their values are unpredictable to attackers. Therefore, normally it is reasonable to assume that the initial states of all registers are uninitialized at the beginning (this assumption was also made by others [69, 24, 87], implicitly or explicitly).

There are two special cases of variable initialization. One case is the PC value, which we always consider to be initialized. The other case is for the instructions whose output is independent of their input. For example, instructions such as "xor eax, eax" or "sub eax, eax" always set eax to zero regardless of the value stored in eax. In this work, we recognize these special instructions and consider the result variables to be initialized. Note that attackers may evade our initialization analysis by crafting certain sequence of instructions to initialize a variable to desired constant without being detected. We will discuss this limitation as well as the solution in Section 4.4.

4.3.3 Detection of Plain Exploit Code

So far, we have described how we prevent attackers from exploiting either self-modifying or indirect jump obfuscation techniques. In this section, we detect the plain exploit code *with or without other obfuscation*, based on system calls and function calls detection. In order to access system resources, the plain exploit code has to use system calls or function calls. In modern operating systems such as Windows and Linux, the system call is the only way to transit from user-mode to kernel-mode to execute privileged instructions. In order to talk with kernel, the plain exploit code has to use either system calls or user-mode APIs (which eventually use system calls). Hence, if we find code patterns of system calls or function calls to user-mode API in an instruction sequence, we consider the instruction sequence as plain exploit code. Previous work [24, 87] also uses a similar idea to detect exploit code. However, their schemes are vulnerable to metamorphic obfuscation [87]. Our approach is robust to most obfuscation techniques such as metamorphism by using static taint and initialization analyses.

Windows and Linux expose system call interfaces through interrupt "int 0x2e" and"int 0x80h" , respectively. Newer versions of Windows and Linux are capable of using optimized "sysenter" instruction. Because the length of these three instructions is only two bytes, even normal network streams may contain plenty of these byte values. Therefore, using only these instructions as a detection criterion will cause a high false positive rate. We observed that before these system call instructions normally several instructions are used to transfer parameters. System call number is an obligatory parameter for a system call, which is stored in the eax register. In addition, most system calls need at least one additional input parameter stored in the ebx register. In fact, only 15 of all 190 system calls do not require the parameter in Linux. Since most of these 15 system calls such as sys_getpid and sys_getuid are merely used to read system or process information in Linux, we believe exploit code must use other system calls to achieve its purposes. Accordingly, we detect system call exploit code in the following way. We first check if an instruction sequence contains system call instructions "int 0x2e","int 0x80h" or sysenter. If the instruction sequence contains one of them, we will analyze the instruction sequence by initialization analysis. If the registers eax and ebx are initialized before system call instructions, we conclude that the instruction sequence

is exploit code.

Note that our detection heuristic works not only for plain exploit code but also for metamorphic code. Instruction "int 0x2e","int 0x80h" and sysenter are the only three instructions to invoke system calls in Windows and Linux, thus the only way to obfuscate system calls is to obfuscate the instructions for transferring parameters. However, because the registers eax and ebx need be initialized despite the metamorphism being used, metamorphic system call code cannot evade our detection.

Attackers may use an alternative way to talk with operating system kernel. The only other way is through an existing user-mode API, which eventually invokes system calls, on the target machine. The only other way is through an existing user-mode API on the target Windows machine. Moreover, most exploit code calls multiple user-mode API functions to achieve their purposes. One example is connectback shellcode. It establishes a TCP connection to a remote host and redirects a command interpreter's output and input to and from the allocated TCP connection. It calls at least five different user-mode API functions [78]: one loads the winsock library ws2_32.dll; one creates a socket; one connects to the remote machine; one executes the command interpreter; one exits the parent process. Another example is portbind shellcode [78]. Portbind shellcode listens on a TCP port and waits for an incoming connection. When the connection is established the code then redirects a command interpreter to the client socket. It calls at least six different user-mode API functions: one creates a socket; one binds to a port; one listens on the port; one accepts a client connection; one executes the command interpreter; one exits the parent process.

It is observed that call instructions are used in Windows exploit code to call user-mode API, and several push instructions are used before calls to transfer parameters. The number of push and call pairs can be used as a threshold to determine the presence of plain exploit code. Clearly, the choice of this threshold normally has to make a tradeoff between false positives and false negatives. In STILL, we set the threshold to two. A natural question is: is this threshold-based detection technique subject to obfuscation? Next we will answer this question.

First, attackers may use only one pair of push and call instructions but actually call multiple user-mode API functions by invoking them several times with different

parameters each time. We name the pair of push and call *caller stub*. For each caller stub, there is an execution path starting from it and eventually reaching it again. In our implementation, we exploit the algorithm for creating strongly connected components [31] of the CFG to detect caller stub. A strongly connected component of a directed graph is a maximal set of vertexes, where every pair of vertexes are reachable from each other. If a pair of push and call uses register operands and they are in a strongly connected component, we consider them as a caller stub.

Second, the push and call instructions themselves are also vulnerable to obfuscation. Figure 4.5 shows two examples of call obfuscation. In Figure 4.5(b), the call is obfuscated by push and jump instructions: $I1$ pushes the return address of the original call to the stack; $I2$ transfers control to the original call target *edx*. In Figure 4.5(c), the call is obfuscated by push and ret: $I1$ pushes the return address of the original call to the stack; $I2$ and $I3$ are equivalent to a jump instruction whose target is *edx*. We notice that no matter how attackers obfuscate call instructions, the obfuscation code bears the same semantics as call: it first pushes the return address into the stack and then sets the call target into program counter. Since the return address of a call is an absolute address within an attack payload, the return address is tainted by PC. Therefore, if the top item of the stack at the entry of a jump instruction is tainted, we consider the jump as an obfuscated call. Similarly, if the second top item of the stack at the entry of a ret instruction is tainted, we consider the instruction ret as an obfuscated call.

```
I1:  call edx        I1:  push I4        I1:  push I4
I4:  ...             I2:  jmp edx        I2:  push edx
                     I4:  ...            I3:  ret
                                         I4:  ...

      (a)                  (b)                 (c)
```

Figure 4.5. Call obfuscation examples (a) original call; (b) call obfuscation by push and jump, $I1$ pushes the return address of the original call to the stack, $I2$ transfers control to the original call target *edx*; (c) call obfuscation by push and ret. $I1$ pushes the return address of the original call to the stack. $I2$ and $I3$ are are equivalent to a jump instruction whose target is *edx*.

Attackers may also obfuscate push instructions. For example, by inserting

Table 4.1. Comparison of STILL and Previous Static Analysis Approaches

			STILL	SigFree	[51]	[24]
Plain Exploit code (without any obfuscation)			Yes	Yes	No	Yes
Anti-Signature	Polymorphism	Encryption	Yes	Partial	Partial	Partial
	Metamorphism		Yes	Yes	Partial	Partial
Anti-static-analysis	Anti-disassembly	Junk byte insertion	Yes	Yes	Yes	Yes
		Opaque predicate	Yes	Yes	No	No
		Code overlap	Yes	No	No	No
		Indirect jump	Yes	No	No	No
		Branch function	Yes	No	No	No
	Self-modifying		Yes	No	No	No
	Memory access obfuscation		No	No	No	No
Anti-emualtion			Yes	Yes	Yes	Yes

redundant pop and push instructions attackers may hide the instructions which actually transfer parameters. However, no matter how attackers obfuscate push instructions, eventually the items below return address in the stack are the parameters.

To reduce false positives, we also perform initialization analysis. We observed that the jump target, return address and parameters must be initialized in function call exploit code. Therefore, if these operands are uninitialized, we do not consider them as attacks.

4.4 Security Analysis

In this section, we analyze the strength and limitations of STILL.

4.4.1 Strength

Besides detecting non-obfuscated exploit code, STILL is also robust to most of these obfuscation techniques. Previous sections showed how STILL detects exploit code that uses polymorphic, self-modifying, indirect jump and branch function obfuscation. Next we discuss how STILL handles the other types of obfuscation techniques.

Metamorphism Metamorphic obfuscation techniques obfuscate the behaviors of a program; however, an obfuscated program still bears the same concrete semantic or abstract semantic of the original one. Since our approach is based on the detection of these invariable abstract semantics, it is robust to metamorphic obfuscation.

For example, Figure 4.6(a) and 4.6(b) show two instances of metamorphic decryption routines generated by the CLET [34] engine, which employs metamorphic obfuscation techniques such as register renaming, equivalent instruction replacement and equivalent functionality. Both instances have the abstract semantic of self-modifying, thus they are subject to detection by STILL.

Register renaming is a simple obfuscation technique that reassigns registers in selected program fragments. For example, the instance in Figure 4.6(a) uses register edx to store the address of encryption data and ecx to encrypted/decrypted data, whereas the instance in Figure 4.6(b) reassigns them to ecx and edx respectively. *Equivalent instruction replacement* is an obfuscation technique that replaces consecutive instructions with other instruction sequences that have the same semantics. For example, the instructions from 29 to 30 in Figure 4.6(a) have the same semantic (plus 4) as instructions 20 and 26 in Figure 4.6(b). *Equivalent functionality* is used to replace a sequence of instructions with another sequence of instructions that has the same functionality. For example, the instructions from 09 to 21 in Figure 4.6(a) have the same functionality (decryption) as instructions from 09 to 18 in Figure 4.6(b).

Since static taint analysis and initialization analysis are not bound to specific registers and specific instructions, they are robust to the above obfuscation techniques.

Anti-disassembly is a type of obfuscation techniques that try to confuse traditional disassembly algorithms such as linear sweep [57]. It includes junk byte insertion, opaque predicate, code overlap, indirect jump and branch function obfuscation techniques.

Junk byte insertion in which junk bytes are inserted at locations that are not reachable at run-time to hinder disassembly. This insertion may mislead linear sweep algorithms, but it cannot mislead recursive traversal algorithms [52], which our algorithm is based on.

Opaque predicate is used to transform unconditional jumps into conditional branches. It allows an obfuscator to insert junk bytes either at jump targets or in the place of the fall-through instruction. We note that opaque predicate may make our approach mistakenly interpret junk byte as executable code. However, this mistake will not cause our approach to miss any real malicious instructions.

```
00 : jmp short 00000038
02 : pop edx
03 : xor eax,eax
05 : mov al,68
07 : mov ecx,[ds:edx]
09 : xor ecx,A81DA19E
0f : sub ecx,DE4285CC
15: sub ecx,202FCF59
1b: sub ecx,1D8F8F5D
21: sub ecx,A3293350
27: mov [ds:edx],ecx
29: sub edx,-2
2f: inc edx
30: inc edx
31: sub al,3
33: dec eax
34: je short 0000003D
36: jmp short 00000007
38: call 00000002
.
```

(a)

```
00 : jmp short 00000030
02 : pop ecx
03 : xor eax,eax
05 : mov al,68
07 : mov edx,[ds:ecx]
09 : ror edx,12
0c : xor edx,E54D36FD
12: rol edx,2
15: rol edx,F
18: add edx,2F3AED50
1e: mov [ds:ecx],edx
20: sub ecx,-3
26: inc ecx
27: sub al,1
29: dec eax
2a: dec eax
2b: dec eax
2c: je short 00000035
2e: jmp short 00000007
30: call 00000002
```

(b)

Figure 4.6. Two metamorphic decryption routine instances generated by CLET.

Therefore, our approach is also immune to obfuscation based on opaque predicates.

Code overlap is one obfuscation technique in which several instructions share one or more bytes. Code overlap can also confuse linear sweep algorithms, but it cannot confuse recursive traversal algorithms [69].

Anti-emulation Since our approach is a static analysis approach, it is immune to anti-emulation obfuscation techniques.

Comparison with other static analysis approaches Table 4.1 shows that STILL is more robust to various obfuscation techniques than the other static analysis approaches such as SigFree [87], [51] and [24] are. Previous sections show that the previous static analysis approaches could be thwarted by anti-static-analysis techniques such as self-modifying, indirect jump and branch function. SigFree will also fail to detect polymorphic exploit code if the number of useful instructions in the decryption routine of the code is below a threshold [87], whereas STILL can always detect polymorphic exploit code. STILL can detect not only polymorphic exploit code but also plain exploit code, whereas [51] can only detect polymorphic exploit code.

4.4.2 Limitations

The present version of our tool has a few limitations. First, the current implementation does not handle memory address obfuscation. This limitation can be handled by treating memory access conservatively [26].

Second, attackers may use special ways to initialize a variable to a constant as a counterattack to initialization analysis. For example, the sequence of instructions ($mov\ eax, ebx$; $add\ eax, 0x1$; $sub\ ebx, -0x1$; $xor\ eax, ebx$) shows a special way to initialize eax to 0. This limitation can be handled by combining our initialization analysis with symbolic execution [46]. This combination, however, may cause some performance overhead. Fortunately, this overhead is not always incurred, since initialization analysis is only used to reduce false positives.

Finally, like previous work [51, 24, 87], STILL does not detect return-to-libc attacks which do not contain any code. One way to alleviate return-to-libc attacks is to map (through mmap()) the addresses of shared libraries so that the addresses contain null bytes [1] [33].

4.5 Experimental Results

We implemented a stand-alone and a proxy-based prototype of STILL to evaluate our technique. The stand-alone prototype consists of the following three parts: (1) *loader* to load the input files from disks; (2) *disassembler* to distill instruction sequences from the input; (3) *analyzer* to analyze the instruction sequences. The prototype will raise one of the following four types of detailed warnings for each input file that is identified with exploit code: (Type1) A function call exploit code is detected; (Type2) A system call exploit code is detected; (Type3) An indirect jump obfuscation code is detected; and (Type4) A self-modifying obfuscation code is detected. Note that polymorphic exploit code is included in Type4. The proxy-based prototype is similar except that it operates as a proxy taking its input from the network.

[1] Null bytes, which are C string terminators, cause attacks terminate before they overflow the entire buffer.

4.5.1 Detection Effectiveness

Polymorphic Shellcode To evaluate the detection effectiveness of STILL, we collected 12,000 polymorphic attack messages from 10 publicly available polymorphic engines. Among these ten, seven engines are from the Metasploit framework [10], including Countdown, Alpha2, JumpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShikataGaNai. The other three engines are CLET [34], ADMmutate [59], and JempiScodes [8]. Countdown uses a decrementing byte as the key for decryption; Pex and JumpCallAdditive are xor decoders and use call to get PC; PexFnstenvMov and PexFnstenvSub are xor decoders and use fnstenv to get PC; Alpha2 generates alphanumeric payload; ShikataGaNai, CLET, ADMmutate, and JempiScodes are advanced polymorphic engines, which also obfuscate the decryption routine by metamorphism such as instruction replacement and garbage insertion. CLET also uses spectrum analysis to defeat data mining methods.

For JempiScodes, we generated 3,000 different attack messages, 1,000 per each of its three obfuscation algorithms. We also generate 9,000 different attack messages, 1,000 per each of the other nine engines. We tested the stand-alone prototype of STILL using these 12,000 attack messages. All of these messages are successfully detected and reported as warning Type4. We also use these 12,000 attack messages to test SigFree. SigFree can only detect 4,103 of them, because the other attack messages have only 6 to 10 useful instructions which is much below than the detection threshold 15 used in SigFree. Our result shows that STILL has much better detection coverage than SigFree.

Plain exploit code from Metasploit framework There are totally 23 different Windows plain shellcode and 11 different Linux plain shellcode in Metasploit framework v2.5. We generated 34 attack messages for each of them. Note that for multi-stage shellcode such as "Linux IA32 Staged Bind Shell", we use the loading (first) stage code to generate attack messages to show that we can detect multi-stage shellcode at its first stage. We tested all of them by the stand-alone prototype. All Linux plain shellcodes are detected as Type2. "Windows Execute Command" and "Windows Execute net user" are detected as Type3, and all other Windows plain shellcodes are detected as Type1.

Worms We also tested on worms CodeRed I, CodeRed II, Sasser, Slammer and Blaster. CodeRed I, CodeRed II, Sasser and Slammer are successfully detected as

Type1. Blaster is successfully detected as Type4, since it exploits polymorphism obfuscation.

4.5.2 Detecting Code in the Wild

We evaluated STILL in the wild using packets collected by LEURRE'COM [2]. LEURRE'COM is a world-wide distributed honeypot project, which has more than 50 platforms – each of which consists of multiple honeypot hosts. Our test dataset contained 475,297 incoming UDP/TCP packets collected at a honeypot platform in France from March 22 2007 to April 21 2007. To avoid raising multiple alerts on a large attack message that were fragmented into multiple packets, we also performed packet reassembly.

Compared STILL with Snort We compared STILL with Snort 2.6.1.4 Win32 Build 54 installed with VRT Certified Rules released on April 3 2007 [15]. Besides the default Snort rules, we enabled shellcode.rules, virus.rules, spyware-put.rules, and specific-threats.rules for more comprehensive detection. For both Snort and STILL, we used one system to process the packets detected as malicious by the other. To verify false alarms, we manually analyzed the packets that only one system detected but the other did not. Snort and STILL results overlapped on 2,029 attack messages in the dataset. There were 15 messages only flagged by Snort as attacks, however, 2 of them were false positives and the others were directory traversal attacks which were not code-injection attacks. In the packets that Snort missed, STILL detected 45 attack messages including 6 instances of the same polymorphic engine and 39 messages of 9 different types of plain exploit code. The result shows that STILL can detect some attack messages which Snort cannot detect, because STILL does not rely on signatures.

Compared STILL with SigFree We also compared STILL with SigFree. STILL detected not only all 2068 attacked messages which SigFree could also detect, but also 6 polymorphic attack instances which SigFree missed. We manually analyzed these 6 packets and found that each of them contained only 7 useful instructions, which is much lower than the detection threshold 15 used in SigFree. This result again shows that STILL has much better detection coverage

[2]http://www.leurrecom.org

than SigFree.

We note that the LEURRE'COM deployment is a low interaction deployment and the deployment provides only limited information in terms of collected payloads. Therefore, it is possible that many detected packets correspond to the same categories of attacks. Testing STILL on the new dataset available in the LEURRE'COM but collected with the new Scriptgen based honeypots. is definitely part of the near future work we will do.

4.5.3 False Positives

Legitimate HTTP traffic We tested the prototype over real HTTP traffic shown in Table 4.3(a). Due to privacy concerns, we were unable to deploy a prototype in a large network to examine real-time traffic. To make our test as realistic as possible, we deployed a client-side proxy underneath a web browser. The proxy recorded all the http requests and replies of a user during his/her daily Internet surfing. During a three-week period, 7 people installed the proxy and helped with collecting totally 378,158 HTTP requests/replies stored in 7 separate datasets. The replies and requests include various types of multimedia data such as video/audio and image files, manually typed urls, clicks through various web sites, search results from search engines such as Google and Yahoo, secure logins to email servers and bank servers, and HTTPS requests. The overall size of the traces is about 1.77 GB.

Our detection results are as follows. First, STILL raised no alarms on the HTTP requests to servers. Second, STILL raised warnings on 30 HTTP binary replies coming into client-side web browsers. Because none of them contain malicious exploit code, they all are false positives and the false positive rate is 0.0079%. Table 4.3(b) shows the number of false positives classified by mime-types in HTTP headers of input. We note that although there are 30 false positives, most of them can be easily precluded by further manual or automatic analysis. First, by checking these files, we found that nine of them were indeed Win32 executable in PE (Portable Executable) file format. Because the purpose of STILL is to detect if a request contains code/mailicous code rather than to discern malicious code from legitimate code, we could let end users to decide whether or not to block these

Table 4.2. Datasets and false positives

Name	Requests	Replies	Size(MB)
Dataset1	5,026	3,584	317.24
Dataset2	32,666	27,253	288.53
Dataset3	87,260	52,772	355.99
Dataset4	17,871	16,941	320.91
Dataset5	26,759	28,028	129.41
Dataset6	17,075	10,042	237.52
Dataset7	46,107	6,774	158.28
Total	232,764	145,394	1,807.88

(a) HTTP real-traffic datasets

Mime-type	Warning Type			
	1	2	3	4
application/octetstream	10	1	0	0
application/x-mms-framed	0	1	0	0
application/x-shockwave-flash	0	2	0	0
audio/mpeg	0	1	0	0
image/gif	0	6	0	0
image/jpeg	0	2	0	0
text/plain	0	1	0	0
flv-application/octet-stream	0	0	0	2
video/flv	0	0	0	1
video/x-flv	0	1	0	2
Total	10	15	0	5

(b) False positives classified by mime-types

executables. Second, some Type2 warnings are obviously false positives because register eax is set to a number much larger than 256, which is not a valid system call number in Linux and Windows. Finally, we can reduce false positives by checking the context. For example, if we believe there are no vulnerabilities in a Shockwave Player (because of other reliable tools), we may ignore the two false positives where the mime type is "application/x-shockwave-flash".

4.5.4 Performance Evaluation

The stand-alone prototype was written in C++ programming language in Win32 environment, and compiled with Borland C++ version 5.5.1 at optimization level -O2. The experiments were performed in a Windows XP Professional SP2 environment running on a Pentium CPU operating at 3.0GHz, equipped with 1GB 533MHz DDR2-SDRAM ECC main memory.

We measured the throughput of the stand-alone prototype over the real traffic datasets. The results are summarized in Table 4.3. To remove the overhead specific to the test environment, we loaded the input files into main memory first and measured the time for our detection algorithm to scan the files. Also, the time spent by the loader was deliberately excluded when we measured the elapsed time. The average processing speed was 1.95Mbits/sec; however, 83% of elapsed time was spent on disassembly. Since the *instruction decoder* part of our disassembler, borrowed from Ollydbg debugger [90], was originally used for offline decoding, we believe optimizing the instruction decoder could dramatically increase the overall throughput. We will investigate it in our future work. For the 1.77 GB of input processed, our analyzer processed disassembled sequences at 11.62Mbits/sec.

Table 4.3. Experiment results

Input data	Elapsed time (sec)		Analysis throughput (Mbps)
	Disassembly	Analysis	
Dataset1	855.785467	267.055825	9.503267
Dataset2	1075.931188	179.813182	12.836709
Dataset3	1373.455207	167.464756	17.00598
Dataset4	1047.691866	233.835534	10.978939
Dataset5	483.113477	82.381132	12.567039
Dataset6	825.120756	228.234238	8.325504
Dataset7	493.956068	85.463046	14.816603

Performance Optimization There are several possible ways to optimize the performance of STILL. One way is that we need not check the requests to the applications that are believed to be vulnerability-free, thus reducing processing time. For example, as we mentioned, if we believe a Shockwave Player has no vulnerabilities, we need not check the HTTP replies whose mime types are "application/x-shockwave-flash". Another way is to exploit some exploit code restrictions to

improve performance [69]. Because most applications treat their input as a string, they will truncate all bytes after the first NULL character of the input. Therefore, one restriction of exploit code is that it normally does not contain any NULL characters. Based on this restriction we only need check the input before the first NULL character. Note that in rare cases some applications accept NULL characters as a part of the input. In these cases, the performance optimization should be turned off.

4.6 Conclusion

We developed STILL, a novel static taint and initialization analysis approach, to address code obfuscation. Our static taint analysis technique enables STILL to collect strong evidence of self-modifying and/or indirect jump code obfuscation behaviors. As a result, while self-modifying code or indirect jumps may avoid the detection of other static analysis approaches, STILL can detect them with a high accuracy. Our experiment results show that STILL detected all the 12,000 exploit codes generated by 10 well-known shellcode generation engines and STILL achieves 0.0079% false positive rate in analyzing 378,158 HTTP requests/replies.

Chapter 5

Bahavior Based Software Theft Detection

5.1 Introduction

Software theft is an act of reusing someone else's code, in whole or in part, into one's own program in a way violating the terms of original license. Along with the rapid developing software industry and the burst of open source projects (e.g., in SourceForge.net there were over 230,000 registered open source projects as of Feb.2009), software theft has become a very serious concern to honest software companies and open source communities. As one example, in 2005 it was determined in a federal court trial that IBM should pay an independent software vendor Compuware $140 million to license its software and $260 million to purchase its services[1] because it was discovered that certain IBM products contained code from Compuware.

To protect software from theft, Collberg and Thoborson [28] proposed software watermark techniques. Software watermark is a unique identifier embedded in the protected software, which is hard to remove but easy to verify. However, most of commercial and open source software do not have software watermarks embedded. On the other hand, "a sufficiently determined attackers will eventually be able to defeat any watermark." [27]. As such, a new kind of software protection techniques called software birthmark were recently proposed [80, 74, 62, 81]. A

software birthmark is a unique characteristic that a program possesses and can be used to identify the program. Software birthmarks can be classified as static birthmarks and dynamic birthmarks.

Though some initial research has been done on software birthmarks, existing schemes are still limited to meet the following five highly desired requirements: (R1) Resiliency to semantics-preserving obfuscation techniques[29]; (R2) Capability to detect theft of components, which may be only a small part of the original program; (R3) Scalability to detect large-scale commercial or open source software theft; (R4) Applicability to binary executables, because the source code of a suspected software product often cannot be obtained until some strong evidences are collected; (R5) Independence to platforms such as operating systems and program languages. To see the limitations of the existing detection schemes with respect to these five requirements, let us break them down into four classes: (C1) static source code based birthmark [80]; (C2) static executable code based birthmark [63]; (C3) dynamic WPP based birthmark [62]; (C4) dynamic API based birthmark [74, 81]. We may briefly summarize their limitations as follows. First, Classes C1, C2 and C3 cannot satisfy the requirement R1 because they are vulnerable to simple semantics-preserving obfuscation techniques such as outlining and ordering transformation. Second, C2, C3 and C4 detect only whole program theft and thus cannot satisfy R2. Third, C1 cannot meet R4 because it has to access source code. Fourth, existing C3 and C4 schemes cannot satisfy R5 because they rely on specific features of Windows OS or Java platform. Finally, none of the four class schemes has evaluated their schemes on large-scale programs.

In this chapter, we propose behavior based birthmarks to meet these five key requirements. Behavior characteristics have been widely used to identify and separate malware from benign programs[47, 25]. While two independently developed software for the same purpose share many common behaviors, one usually contains unique behaviors compared to the other due to different features or different implementations. For example, HTML layout engine Gecko engine [6] supports MathML, while another engine KHTML [9] does not; Gecko engine implements RDF (resource description framework) to manage resources, while KHTML engine implements its own framework. The unique behaviors can be used as birthmarks for software theft detection. Note that we aim to protect large-scale software.

Small programs or components, which may not contain unique behaviors, are out of the scope of this dissertation.

A system call dependence graph (SCDG), a graph representation of the behaviors of a program, is a good candidate for behavior based birthmarks. In a SCDG, system calls are represented by vertices, and data and control dependences between system calls by edges. A SCDG shows the *interaction* between a program and its operating system and the interaction is an essential behavior characteristic of the program [47, 25]. Although a code stealer may apply compiler optimization techniques or sophisticated semantic-preserving transformation on a program to disguise original code, these techniques usually do not change the SCDGs. It is also difficult to avoid system calls, because a system call is the only way for a user mode program to request kernel services in modern operating systems. For example, in operating systems such as Unix/Linux, there is no way to go through the file access control enforcement other than invoking open()/read()/write() system calls. Although an exceptionally sedulous and creative plagiarist may correctly overhaul the SCDGs, the cost is probably higher than rewriting his own code, which conflicts with the intention of software theft. After all, software theft aims at code reuse with disguises, which requires much less effort than writing one's own code.

We develop system call dependence graph (SCDG) birthmarks for meeting these five key requirements. To extract SCDG birthmarks, automated dynamic analysis is performed on both plaintiff and suspect programs to record system call traces and dependence relation between system calls. Since system calls are low (abstraction) level implementation of interactions between a program and an OS, it is possible that two different system call traces represent the same behavior. Thus, we filter out noises, which cause the difference, from system call traces in the second step. Then, SCDGs are constructed and both plaintiff and suspect SCDG birthmarks are extracted from the SCDGs.

We exploit subgraph isomorphism algorithm to compare plaintiff and suspect SCDG birthmarks. Although subgraph isomorphism is NP-complete in general, it is tractable in this application. First, the size of SCDGs is limited by a predefined parameter (100 or 400 in our experiment). Second, SCDGs are not general graphs. Their characteristics such as various types of nodes, makes backtrack-based iso-

morphism algorithm efficient. Finally, the first matching suffices for software theft detection, whereas the traditional isomorphism testing finds all isomorphism pairs. Hence, the isomorphism testing on SCDGs is tractable and efficient in practice.

We makes the following contributions:

- We proposed a new type of birthmarks, which exploit SCDGs to represent unique behaviors of a program. Without requiring any source code from the suspect, SCDG birthmark based detection is a practical solution for reducing plaintiff's risks of false accusation before filing a lawsuit related to intellectual property.
- As one of the most fundamental runtime indicators of program behaviors, the proposed system call birthmarks are resilient to various obfuscation techniques. Our experiment results indicate that they not only are resilient to evasion techniques based on different compilers or different compiler optimization levels, but also successfully discriminates code obfuscated by two state-of-the-art obfuscators.
- We design and implement Hawk, a dynamic analysis tool for generating system call traces and SCDGs. Hawk potentially has many other applications such as behavior based malware analysis. Detailed design and implementation of Hawk are present in this chapter
- To our knowledge, SCDG birthmarks are the first birthmarks which are proposed to detect software component theft. Moreover, unlike existing works that are evaluated with small or toy software, we evaluate our birthmark on a set of large software, for example web browsers. Our evaluation shows the SCDG birthmark is a practical and effective birthmark.

The rest of this chapter is organized as follows. Section 5.2 introduces concepts and measurements about software birthmarks. In Section 5.3, we propose the design of SCDG birthmarks based software theft detection system. Section 5.4 presents evaluation results. We discuss limitation and future work in Section 5.5. Finally, we draw our conclusion in Section 5.6.

5.2 Problem Formalization

This section first presents the definitions related to software birthmark and SCDG birthmark, and then introduces the metric to compare two SCDG birthmarks.

5.2.1 Software Birthmarks

A software birthmark is a unique characteristic that a program possesses and that can be used to identify the program. Before we formally define software birthmarks, we first define the meaning of *copy*. We say a program q is a copy of program p if q is exactly the same as p. Beyond that, q is still considered as a copy of p if it is the result of applying semantic preserving transformation (e.g., obfuscation techniques and compiler optimization) over p. The following definition of software birthmark and dynamic software birthmark is a restatement of the definition by Tamada et al. [80] and Myles et al. [62]:

Definition 4. *(Software Birthmark) Let p, q be programs or program components. Let $f(p)$ be a set of characteristics extracted from p. We say $f(p)$ is a birthmark of p, only if both of the following conditions are satisfied:*

- *$f(p)$ is obtained only from p itself.*
- *program q is a copy of $p \Rightarrow f(p) = f(q)$.*

Software birthmarks can be classified as static birthmarks and dynamic birthmarks. A static birthmark relies on syntactic structure of a program. Existing static birthmarks are vulnerable to simple semantic-preserving transformations [62]. On the other hand, a dynamic birthmark relies on runtime behavior of a program, which is more difficult to be altered through code obfuscation techniques. The system call based birthmark we propose in this chapter is a dynamic birthmark.

Definition 5. *(Dynamic Software Birthmark) Let p, q be programs or program components. Let I be an input to p and q. Let $f(p, I)$ be a set of characteristics extracted from p when executing p with input I. We say $f(p, I)$ is a dynamic birthmark of p, only if both of the following conditions are satisfied:*

- *$f(p, I)$ is obtained only from p itself when executing P with input I.*
- *program q is a copy of $q \Rightarrow f(p, I) = f(q, I)$*

5.2.2 SCDG Birthmarks

Before we define SCDG birthmarks, we first define SCDG. A system call dependence graph (SCDG) is a graph representation of the behaviors of a program,

where system calls are represented by vertices, and data and control dependences between system calls are represented by edges. To formally define SCDG, we first define dynamic dependence graph (DDG) of a program run. Later, an SCDG can be constructed from a DDG.

Definition 6. *(DDG: Dynamic Dependence Graph) The dynamic dependence graph of a program run is a 3-tuple graph* $DDG = (N, E, \beta)$*, where*

- N *is a set of nodes, and node* $n \in N$ *corresponds to an executed statement in the program run*
- $E \subseteq N \times N$ *is the set of dependence edges, and each edge* $n_1 \rightarrow n_2 \in E$ *corresponds to a dynamic data dependence or dynamic control dependence between statements* n_1 *and* n_2*. A data dependence exists between two executed statements if the set values used by one statement overlaps the set of value defined by the other. A control dependence is introduced if the execution of one statement depends on the values defined by the other statement, usually a predicate statement.*
- $\beta : E \rightarrow T$ *is a function assigning dynamic dependence types, either data or control, to edges.*

Definition 7. *(SCDG: System Call Dependence Graph) The system call dependence graph of a program run is a 4-tuple graph* $SCDG = (N, E, \alpha, \beta)$*, where*

- N *is a set of nodes, where node* $n \in N$ *corresponds to a system call execution in the program run*
- $E \subseteq N \times N$ *is the set of dependence edges, and each edge* $n_1 \rightarrow n_2 \in E$ *denotes that there exists a dependence path from* n_1 *to* n_2 *in the DDG of the program run.*
- $\alpha : N \rightarrow S$ *is a function assigning system call to nodes,*
- $\beta : E \rightarrow T$ *is a function assigning dependence types, either data or control, to edges.*

Figure 5.2.2 presents an example to illustrate DDG and SCDG. Data and control dependences are plotted in solid and dashed lines, respectively. In Figure 5.2.2 (b), statement executions 2, 10 and 14 have data dependence on statement 1

because they use the file descriptor defined at 1. In Figure 5.2.2 (c), system call execution "7:lseek" has data dependence on "5:read" because of the dependence path $7 \rightarrow 6 \rightarrow 5$; system call execution "12:write" has data dependence on "10:read" due to the dependence path $12 \rightarrow 11 \rightarrow 10$; System call execution "4:open" has control dependence on "2:read" because statement 4 has control dependence on 3 and 3 has data dependence on 2 .

Next, we define subgraph isomorphism which will be used to compare similarity of SCDGs.

Definition 8. *(Graph Isomorphism) A bijective function $f : N \rightarrow N'$ is a graph isomorphism from a graph $G = (N, E, \alpha, \beta)$ to a graph $G' = (N', E', \alpha', \beta')$ if*

- $\forall n \in N, \alpha(n) = \alpha(f(n))$,
- $\forall e = (n_1, n_2) \in E, \exists e' = (f(n_1), f(n_2)) \in E'$ *such that* $\beta(e) = \beta(e')$,
- $\forall e' = (n'_1, n'_2) \in E', \exists e = (f^{-1}(n'_1), f^{-1}(n'_2)) \in E$ *such that* $\beta(e') = \beta(e)$

Definition 9. *(Subgraph Isomorphism) A bijective function $f : N \rightarrow N'$ is a subgraph isomorphism from a graph $G = (N, E, \alpha, \beta)$ to a graph $G' = (N', E', \alpha', \beta')$ if there exists a subgraph $S \subset G'$ such that f is a graph isomorphism from G to S.*

Definition 10. *(γ-Isomorphism) A graph G is γ-isomorphic to G' if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G', and $|S| \geq \gamma|G|$, $\gamma \in (0, 1]$.*

Definition 11. *(SCDGB: System Call Dependence Graph Birthmark) Let p be a program or program component. Let I be an input to p, and $SCDG_p$ be system call dependence graph of the program run with input I. SCDG birthmark $SCDGB_p$ is the subgraph of the graph $SCDG_p$ that satisfies the following conditions:*

- *program or component q is a copy of p and $SCDG_q$ be system call dependence graph of the program run of q with input $I \Rightarrow SCDGB_p$ is subgraph isomorphic to $SCDG_q$.*
- *program or component q is different from p and $SCDG_q$ be system call dependence graph of the program run of q with input $I \Rightarrow SCDGB_p$ is not subgraph isomorphic to $SCDG_q$.*

According to our definition of SCDG birthmark, program q is regarded as plagiarized from program p if the SCDG birthmark of p is subgraph isomorphic to SCDG of q. Although as shown in our experiment SCDG birthmark is robust to state-of-the-art obfuscation techniques, for robustness to unobserved and unexpected attacks, we relax subgraph isomorphism to γ-isomorphism in our detection. q is regarded as plagiarized from that of p, if the SCDG birthmark of p is γ-isomorphic to SCDG of q. We set γ 0.9 in experiments because we believe that overhauling 10% of a SCDG birthmark is almost equivalent to changing the behavior of a program.

5.3 System Design

5.3.1 System Overview

Figure 5.2 shows the overview of our system. It consists of four stages: dynamic analysis, noise filtering, SCDG birthmark extraction, and birthmark comparison. Let us summarize each of the steps before dealing with details in later subsections.

Dynamic Analysis. In the first step, automated dynamic analysis is performed on both plaintiff and suspect programs to record their system call traces. For both programs, we feed in the same input. Besides system calls, the call stack for each system call and the dependence relation among system calls are calculated and recorded.

Noise Filtering. System calls are low level implementation of interactions between a program and an OS. It is possible that two different system call traces represent the same behavior, e.g., because of the existence of many system calls that are dependent on runtime environment. Therefore, we filter out noises from system call traces before extracting birthmarks.

SCDG Birthmarks Extraction. We aim to detect component theft. Therefore, in this step, we first identify the system calls invoked by the component of interest in a plaintiff program and then extract SCDGs for the component. Then, we divide SCDGs of the component into subgraphs, and refine the subgraphs by removing common nodes that are also found in SCDGs of several unrelated programs. Finally, the remaining subgraphs are considered as birthmarks of the

plaintiff component.

Although it is possible to choose the SCDG of the whole suspect program for comparison, the graph's size would be too large to efficiently test subgraph isomorphism. As such, we also divide the SCDG of a suspect program into subgraphs.

Birthmark Comparison. Once both plaintiff and suspect birthmarks are extracted, we examine the birthmarks for a r-isomorphism using relaxed VF subgraph isomorphism algorithm [40]. To increase the efficiency, three forms of pruning are performed to reduce the search space.

5.3.2 Dynamic analysis

In this subsection, we first briefly introduce our dynamic analysis system. Then, we describe the design details of the dynamic instrumentation. Finally, deferred reference counting is introduced and discussed to improve performance.

Our dynamic analysis system consists of Valgrind [64] and Hawk, as shown in Figure 5.3. Valgrind is a generic framework to instrument machine code at runtime, and Hawk is a plugin tool we designed and developed for Valgrind. Valgrind and Hawk work together to generate system call traces and their dependences. Specifically, Valgrind takes a binary client program, which is a plaintiff or a suspect program in our case, and an input to the client program for dynamic analysis. Then, it decompiles the client's machine code, one small code block at a time, in a just-in-time, execution-driven fashion. It disassembles the code block into an architecture-neutral intermediate representation (IR) block. In Hawk, every memory byte and register of the client program is *shadowed* by a dependence set, which is a set of system calls it depended on. Hawk instruments the IR block given by Valgrind with analysis code. The analysis code is used to update the shadow values of the client program's memory locations and registers. Then, the instrumented IR block is converted back into machine code by Valgrind and executed. The resulting translation is stored in a code cache and thus it can be reused without calling the instrumenter again. Valgrind also provides system call hooks for Hawk to instrument system calls of client programs. When a system call of a client program is invoked, Hawk create a new node for the system call as well as the dependence edges between the new node and the other ones.

Next, we presents the details of IR instrumentation. The Valgrind IR majorly consists of five types: load memory IR, store memory IR, get register IR, put register IR, and expression IR. The first four types of IR are used to read or write values from memory and registers to the temporary variables of an IR block. In Hawk, the instrumentation to the first four types of IR is just transferring of shadow values between a temporary variable and a register/memory location. All actual operations are performed by expression IR. An expression IR is abstracted as $t_d = op(t_1, t_2, ..., t_n)$ (n¡=4 in practice), where $t_1, t_2, ..., t_n$ denote the set of temporary variables used by the IR and t_d denotes the temporary variable defined. The instrumentation of the expression IR is defined by $sh(t_d) = \cup(sh(t_1), sh(t_2), ..., sh(t_n)$. Figure 5.4 shows an example of an IR block and its instrumentation.

Besides IR instrumentation, Hawk also instruments the system calls of a client program to create new system call nodes and dependence edges. When a system call occurs, a handler function of Hawk is called. The system call information (number, index, parameter and result) and its call stack is recorded within the function. In addition, a new system call node is created, and dependence edges between the new system call node and previous nodes are established by the shadow values of the system call's input parameters. For example, the *read* system call in X86 Linux uses register *ebx* as the input parameter, which stores a file descriptor id. Dependence edges are created between this *read* system call node and the nodes in the shadow values of ebx. Finally, the new system call node is assigned to the shadow variables of the system call's output parameters. For example, the *eax* register is the return variable of the *open* system call, storing the descriptor id of an opened file . The newly created *open* system call node is assigned to the shadow value of the *eax* register.

For large programs, Hawk may generate a great number of intermediate dependence sets. Thus, a garbage collector for dependence sets is needed. There are several ways to implement a garbage collector, such as reference counting, mark-sweep and copy collection. Here we use reference counting instead of mark-sweep or copy collection because the number of dependence sets during execution are huge and tracing would be prohibitively slow. Also, there are no cycles to cause problems. However, a disadvantage of reference counting is that frequent update

of reference count may hurt performance. This is a severe problem in our case, because every instrumentation of an IR may need to update reference count. To solve this problem, we exploit deferred reference counting [18]. Deferred reference counting was originally used to reduce the cost of maintaining reference counts by avoiding adjustments when the reference is stored in the stack. In our case, we avoid updating references on temporal variables due to the short lifetime of the temporal variables. During the execution, dependence sets cannot be reclaimed as soon as their reference counts become zero. Because there might still be references to them from temporal variables, such sets are added to a zero count table (ZCT) instead. The dependence sets in the ZCT are scanned at the end of code blocks, and any sets with zero reference count are reclaimed.

Currently, Hawk does not trace control dependence for efficiency concerns. Our experiments in section 5.4 show that data dependence alone is powerful enough for software theft detection.

5.3.3 Noise Filtering

As a low level abstraction of the interaction between a program and the OS, system call sequences contain noise. Due to the noise system calls, the same behavior could be represented by two different system call sequences. We filter out the noises from system call traces in the following ways. First, some types of system calls are ignored because they apparently do not represent the behavior characteristic of a program. For example, the system call *gettimeofday* returns the elapsed time since Epoch in seconds and microseconds. Many programs periodically call *gettimeofday* with no significant impact on their behaviors; therefore, we remove *gettimeofday* if no other system calls depend on them. Another example is related to memory management system calls. A libc *malloc* function is normally implemented by system call *brk* and/or *mmap*. The *mmap* system call is used when extremely large segments are allocated, while the *brk* system call changes the size of the heap to be larger or smaller as needed. Normally, C function *malloc* first grabs a large chunk of memory and then splits it as needed to get smaller chunks. As such, not every malloc in C involves a system call and two identical programs may have very different memory management system call sequences. Therefore, we

ignore all memory management system calls. Second, some types of system calls are considered as the same in system call birthmarks, because some system calls provide multiple versions that take slightly different parameters for convenience. For example, *fstat(int fd, struct stat *sb)* system call is the same as *stat(const char *path, struct stat *sb)* except that *fstat* uses the file descriptor *fd* as its parameter instead of the file name *path*. By considering such system calls identical, we can not only reduce the sophistication of dealing with many different system calls, but also address the counterattack where an attacker replaces one system call with another. Finally, since failed system calls do not affect the behavior characteristic of a program, they are also ignored. For example, when a program tries to open a file, it may fail at the first time but succeed at the second time. Although system call *open* is called twice, here the first failed call should be removed.

5.3.4 Extraction of SCDG Birthmarks

⊙ *Extraction of Plaintiff Birthmarks.* There are four steps to extract SCDG birthmarks for a plaintiff component. First, by analyzing the system call trace whose noise has been removed, we determine whether a system call is called by a plaintiff component. This is useful for detecting software component theft because we need to know which system calls are invoked from which component of a plaintiff program. Specifically, there are two implementation options. One method is to use a special list, L, containing information on the functions belonging to the plaintiff component. List L can be automatically generated by processing the source code of the plaintiff component with a tool such as Elsa[5]. Then, whenever a system call is called, Hawk can notice whether the system call is called by the plaintiff component by searching in the call stack (containing callers of the system call) for any occurrence of the functions listed in L. Note that we can preserve the symbol table of the plaintiff component because we have control over the compilation of plaintiff source codes. Alternatively, a simpler method which does not need to maintain list L is available. If we can compile the plaintiff component into a dynamic linked library (DLL), Hawk can simply use a utility function provided in Valgrind to retrieve the DLL component that contains any of the callers of the invoked system call.

Second, an SCDG and a dynamic call tree are built from the system call traces corresponding to the the plaintiff component. Building an SCDG is trivial given nodes (system calls) and edges (dependences). Besides SCDG, we also build a dynamic call tree, which will be used to partition an SCDG in later steps. A dynamic call tree here is a tree with subroutine calls as internal nodes and system calls as leaf nodes. A node's parent is its caller and its children are its callees. The path from a leaf system call node to the root node is the call stack of the system call. The process of generating a dynamic call tree is also trivial: we just need to merge all the call stacks.

Third, we divide an SCDG into subgraphs. Because an extracted SCDG may be too large to efficiently compute subgraph isomorphism and/or too specific for the plaintiff program, they are not directly used as birthmarks. As a component theft is mostly likely to happen over a subroutine, we partition the graph based on dynamic call tree. That is, we extract a subtree from the dynamic call tree, and the leaf nodes in the subtree and their dependence relation consist of an SCDG birthmark candidate. The partition process is as the following. We first set a parameter m, which is used to guarantee that the subgraph is not too large or too specific for the partition. Then, the dynamic call tree of the selected component is traversed by a depth first search algorithm. When a tree node is visited, the number of leaf nodes in this subtree is calculated. If the number is less than m, the subtree is selected and search within the subtree is skipped. The process is finished when all noes in the dynamic call tree is traversed.

Finally, we remove the SCDG subgraphs which represent common behaviors. This step is necessary because we need to find the unique behavior of plaintiff components as birthmarks. For this purpose, a set of training programs are used. The set of programs should include programs which have a similar component with the plaintiff program but are indeed completely unrelated programs. The SCDG subgraphs are compared with the SCDGs of the training set. All SCDG subgraphs which are subgraph isomorphism to the SCDGs of the training set are removed, and finally, the remaining subgraphs become birthmarks.

⊙ *Extraction of Suspect Birthmarks.* As mentioned earlier, we assume that there are no source code and symbolic debugging information of a suspect program. Hence, it is difficult to identify the suspicious components in a suspect program,

not to mention extracting SCDGs from them. Thus, we have to extract SCDG birthmarks based on the SCDG of the whole suspect program. Specifically, we partition the whole SCDG according to dynamic call tree, as in the case for plaintiff birthmark extraction, except that we choose m to be several times larger.

5.3.5 Birthmark Comparison

Once both the plaintiff and suspect SCDG birthmarks are extracted, they are compared using (relaxed) VF subgraph isomorphism algorithm [40]. $n * m$ pairs subgraph isomorphism testing are needed in principle, where n and m are the numbers of plaintiff and suspect birthmarks, respectively. Fortunately, most pairs can be pruned through three forms of loseless pruning.

Pruning Search Space First, SCDG birthmarks smaller than an interesting size K or the types of system calls smaller than T are excluded from both plaintiff and suspect birthmarks. For software theft detection, we only need to locate birthmark pairs of non-trivial ones, which, if found, can provide strong evidence for proving theft. Second, based on the definition of $\gamma - isomorphism$, a SCDG birthmark pair (g, g') can be excluded if $|g'| < \gamma|g|$, where g and g' are SCDG birthmarks of plaintiff and suspect programs, respectively. Finally, a SCDG birthmark pair can be pruned based on the characteristics of SCDGs. In this chapter, we use vertex histogram as the characteristics of SCDGs and the similarity between two vertex histograms can be used for pruning. Specifically, a plaintiff SCDG birthmark g is represented by vertex histogram $h(g) = (n_1, n_2, ..., n_k)$, where n_i is the frequency of the ith kind of vertices, and a suspect SCDG birthmark g' is represented by $h(g') = (m_1, m_2, ..., m_k)$. We define the difference between $h(g)$ and $h(g')$ as $d(h(g), h(g')) = \sum_{i=0}^{k} e_i$, where $e_i = n_i - m_i$ if $n_i > m_i$ and $e_i = 0$ if $n_i \leq m_i$. Based on the definition of $\gamma - isomorphism$, the pair (g, g') can be excluded if $d < (1 - \gamma)|g|$.

Computational Feasibility. Because our SCDG birthmark involves subgraph isomorphism testing, we discuss the computation feasibility of the testing. As mentioned in [58], although subgraph isomorphism is NP-complete in general, research in the past three decades has shown that some algorithms are reasonably fast on average and become computationally intractable in a few cases [39, 38].

For instance, algorithms based on backtracking and look-ahead, e.g., Ullmann's algorithm [84] and VF [40], are suitable with graphs of thousands of nodes.

In addition to the general tractability issue, the characteristics of graphs and the needs for a specific application also reduce the computational cost [58]. In our application, the computational cost are reduced for the following reasons. First, the size of SCDGs is limited by a predefined parameter (100 or 400 in our experiment). Second, SCDGs are not ordinary graphs. Their characteristics such as various types of nodes, makes backtrack-based isomorphism algorithm efficient. Last, the first matching suffices for software theft detection, whereas the traditional testing finds all isomorphism pairs. Hence, the isomorphism testing on SCDGs is tractable and efficient in practice.

Finally, our search space pruning phase can effectively identify and discard the spurious SCDG pairs which are obviously not isomorphic to each other, avoiding detailed isomorphism testing. As a result, only a small portion of SCDG pairs are really tested in the case of a real software theft. Thus, our detection is computationally efficient in our specific problem settings.

5.4 Evaluation

In Section 1 we mentioned five key requirements on software theft detection. It is easy to see R4 and R5 are already met by our design. In this section, we evaluate the performance of SCDG birthmarks with respect to three primary criteria: (M1) capability to detect component theft for large-scale programs, (M2) credibility to independently developed program, and (M3) resiliency to obfuscation. These three criteria contain more than R1, R2 and R3 because of M2.

In the following, we first discuss the implementation of our system and environmental setup. Then, we evaluate criteria M1 and M2 for SCDG birthmarks with over 30 real-world large application programs. Finally, we evaluate criteria M3 against evasion techniques that apply different compilers, different compiler optimization levels, or obfuscation techniques.

5.4.1 Implementation and Environmental Setup

We implemented a prototype of SCDG birthmark based software theft detection system. The entire system consists of about 5,000 lines of C/C++ code and 1000 lines of Tcl script code. Our implementation of the γ-isomorphism testing algorithm was adopted from VFlib[1]. We used tree.hh[2], an STL-like C++ tree class, for dynamic call tree representation and operation. The version of Valgrind we used is 3.3.1. The entire detection system runs under Ubuntu 8.04. For detection purpose, we fed the same input and perform the same operation (spell checking) if applicable; otherwise, an appropriate input and a simple operation was provided. In our experiment, we set γ 0.9, minimal size of SCDG birthmarks K 15, and maximal size of SCDG birthmarks m 100 for plaintiff programs and 400 for training programs and testing sets (i.e., suspect programs), respectively.

5.4.2 Effectiveness of SCDG

We chose two subject program components for experiments: Gecko layout engine for web browser and GNU Aspell spell checker. Before we give details on the subject components' SCDG birthmarks and show the effectiveness, we first introduce the training program set and the testing program set.

Training Program Set. The following programs were part of the training program collection: Dillo, Yudit, Meld, Gimp, Totem, Pdfedit and Dia. Dillo was chosen for its similar web content rendering behavior with Gecko engine. Yudit was chosen for its similar spell checking behavior with Aspell. Others were chosen for general common behaviors. Each training program was executed under our dynamic analysis system and performed a simple operation and then quit. We fed one of our authors' home page url as input to Dillo and quit it after the home page was displayed. The home page html was also fed to Yudit and performed spell checking and quit. For other programs, appropriate input and a simple operation were provided (e.g. giving Gimp a gif file and then adjust color balance) and then quit. Table 5.1 shows the statistics for the SCDGs of the training program set. Note that for the training program set, we have already known that none of them

[1]http://amalfi.dis.unima.it/graph
[2]http://www.aei.mpg.de/ peekas/tree

contains our subject components.

Table 5.1. Training set statistics

Program	Version	Type	SCDG	
			Node #	Edge #
Dillo	0.8.6	Web Browser	2612	1510
Yudit	2.4.1	Text Editor	4576	1023
Meld	1.1.5.1	Diff Viewer	12314	7084
Gimp	2.4.5	Graph Editor	59372	5972
Totem	2.22.1	Media Player	21865	6762
Pdfedit	0.3.2	PDF Editor	8937	4867
Dia	0.96.1	Diagram Drawing	27145	29185

Testing Program Set. We evaluated Gecko SCDG birthmarks and Aspell SCDG birthmarks against 24 large application programs shown in Table 5.2. Each test program was executed under our dynamic analysis system and perform a simple operation and then quit. Again, we fed the home page url as input to all browsers, and performed spell checking if applicable, and then quit after the home page was displayed. We fed the home page html to all word processors, text editors, instant messengers and email clients and performed spell checking if applicable and quit. For other programs, appropriate input and a simple operation were provided and then quit. Table 5.2 shows statistics for the SCDGs of the test program set. Note that for most of programs in the testing sets, we do not have the preknowledge whether they contain Gecko and/or Aspell or not; that is, our test is a blind test.

Experiment of GNU Aspell. In this experiment, we test whether a program in the testing set contained a small software component – GNU Aspell spell checker. GNU Aspell is the standard spell checker for the GNU software system. It can either be used as a component (library) or as an independent spell checker. As a software component, it has been widely used in word processors, document editors, text editors and instant messengers.

We extracted birthmarks of Aspell from a standalone program Aspell 0.60.5. The extracted SCDG graph contains 481 nodes and 659 edges. One SCDG birthmark, shown in Figure 5.5, was generated after compared with SCDGs of the training programs set (i.e., after removing the common SCDGs). The Aspell SCDG birthmark was compared with SCDGs of the programs in the testing set. The

Table 5.2. Testing set statistics

Program	Version	Type	SCDG	
			Node #	Edge #
Flock	2.0.3	Web Browser	21337	9343
Epiphany	2.22.2	Web Browser	16864	9011
Konqueror	3.5.10	Web Browser	11850	5589
Amaya	10	Web Browser	42701	23958
Opera	9.52	Web Browser	58485	21361
Songbird	1.1.2	Web Browser	37103	25547
Galeon	2.0.7	Web Browser	19825	7450
AbiWord	2.4.6	Word Processor	12975	5642
KWord	1.6.3	Word Processor	15408	6630
LyX	1.5.3	Latex Editor	21977	18656
Texmaker	1.6	Latex Editor	6897	3223
Kile	2.0.0	Latex Editor	50937	24615
Gedit	2.22.3	Text Editor	25113	5867
Bluefish	1.0.7	Text Editor	10952	3502
GNU Emacs	22.2.1	Text Editor	14807	4734
Vim	7.1.138	Text Editor	2582	1979
Pidgin	2.5.2	Messenger	10816	8014
Kopete	0.12.7	Messenger	16319	7144
Kmess	1.5	Messenger	10830	6247
GnoCHM	0.9.9	CHM Viewer	21191	8354
Evince	2.22.2	Doc. Viewer	16179	7095
GV	3.6.3	Doc. Viewer	6508	3267
Quod Libet	1.0	Media Player	15839	10725
Evolution	2.22.3	Email Client	13798	6787

result is that five programs, including Opera, Kword, Lyx, Bluefish, Pidgin, contain the Aspell Birthmark. It shows that each of the five programs contain Aspell component, while others not. This result was confirmed by manually checking the programs in the testing set.

Experiment of Gecko Engine. In this experiment, we study SCDG birthmarks using web browsers and their layout engines. A layout engine is a software component that renders web contents (such as HTML, XML, image files, etc.) combined with formatting information (such as CSS, XSL, etc.) onto the display units or printers. It is not only the core components of a web browser, but also used by other applications that require the rendering (and editing) of web contents.

Gecko [6], which is the second most popular layout engine on the Web, is an layout engine used in most Mozilla software and its derivatives.

We extracted Gecko SCDG birthmarks from Firefox 3.0.4. The extracted SCDG graph contains 726 nodes and 1048 edges. Two SCDG birthmarks were extracted after comparing with the training program set. Figure 5.6 shows an example SCDG birthmark of Gecko. The two Gekco SCDG birthmarks were compared with SCDGs of testing programs set. The result is that four programs, including Flock, Epiphany, SongBird and Geleon, contain one of the two Gecko Birthmarks. It shows that each of the four programs contain Gecko components, while others not. This result was confirmed by manually checking the programs of the testing set.

5.4.3 Impact of Compiler Optimization Levels

Changing compiler optimization levels is a type of semantic preserving transformation techniques which may be used by a software plagiarist to avoid detection. Here, we evaluated the impact of compiler optimization levels on system call based birthmarks. A set of programs were used: bzip2 (the second-most popular lossless compression tool for Linux), gzip (a lossless compression tool) and oggenc (a command-line encoding tool for Ogg Vorbis, a non-proprietary lossy audio compression scheme). To avoid incompatible compiler features, single compilation-unit source code (bzip2.c, gzip.c and oggenc.c) were used [3]. We used five optimization switches (-O0,-O1,-O2,-O3 and -Os) of GCC to generate executables of different optimization levels (e.g., bzip2-O0, bzip2-O3, etc.) for each program. The generated executables were executed with the same input, a system call trace was recorded, and finally SCDGs were generated for each executable. We compared the system call sequences and found that applying optimization options does not change the system call traces and SCDGs of bzip2 and gzip, while the system call traces for oggenc with optimization options (-O3 and -Os) contain only one "write" system call less than that with optimization options (-O0, -O1 and -O2). This result shows that system call based SCDG birthmarks are robust to compiler optimization.

[3] http://people.csail.mit.edu/smcc/projects/single-file-programs/

5.4.4 Impact of Different Compilers

A software plagiarist may also use a different compiler to avoid detection. To evaluate the impact of applying different compilers, we compared system call sequences with three compilers: GCC, TCC and Watcom. We used the three compilers to generate executables for each of the three programs (e.g., bzip2-gcc, bzip2-tcc) we used before. The generated executables were executed with the same input and a system call trace and SCDG is recorded for the each executable. We used GCC result to compare with TCC and Watcom results. The results show that the system call traces and SCDGs are exactly the same between TCC and GCC (both with default optimization levels). The system call traces between GCC and Watcom look different. By checking the compilers, we found that the differences were caused by different standard C libraries used by the compilers, not because of the compilers themselves. Both GCC and TCC use *glibc*, while Watcom uses its own implementation. Three types of differences are found. First, different but equivalent system calls are used between the two libc implementation such as *stat* and *stat64*. Second, failed system calls appear many times in one result, but not in the other. Last, some differences caused by memory management system calls. Fortunately, as mentioned in Section 5.3.3, such differences can be removed by our *noise filtering* step. As such, our proposed birthmarks can survive under the three different compilers in this experiment.

5.4.5 Impact of Obfuscation Techniques

Obfuscation is another type of semantic preserving transformation techniques. There are two types of obfuscation tools: source code based and binary based. A source code obfuscator accepts a program source file and generates another functionally equivalent source file which is much harder to understand or to reverse-engineer. A binary obfuscator exploits binary rewriting technique for obfuscation. We evaluated the impacts of obfuscation techniques over two state-of-the-art obfuscation tools. For source code obfuscation, we used the commercial product *Semantic Designs Inc's* C obfuscator that implements abstract syntax tree (AST) based code transformation. Its features include (but not limited to) identifier scrambling, format scrambling, loop rewriting, and if-then-else rewriting. For bi-

nary code obfucation, we used control flow flattening implemented in Loco based on Diablo link-time optimizer [60]. Control flow flattening can transform statements 's1; s2;' into 'i=1; while(i) { switch(i) { case 1: s1; i++; break; case 2: s2; i=0; break;}}'. We used the two obfuscators to obfuscate and then compile each of the three programs we used before. The generated executables were executed with the same input and a system call trace and SCDG were recorded for the each executable. The system call traces and SCDGs between the original programs and obfuscated programs are exactly the same. This results show that SCDG birthmark is robust to these two state-of-the-art obfuscation tools.

5.5 Discussion

5.5.1 Robustness

Besides those tested obfuscations in Section 5.4.5, SCDG birthmark is robust to most types of obfuscation techniques presented in [29] including split or merge variables, promote scalars to objects, convert static data to procedure, change encoding, change variable life, split or merge arrays, reorder arrays, unroll loop, reorder expression, extend loop condition, reorder statements and so on. Although they can significantly alter the source code and binary code appearance, they do not change system calls and dependence relations between system calls. As a consequence, SCDG birthmark is robust to the all these obfuscations.

5.5.2 Counterattacks

One of the possible counterattacks to SCDG birthmark is the *system call injection attack*. An attacker may insert a great number of system calls in the plagiarism program without compromising its original behaviors. This counterattack may avoid our detection when a small m is used for generating suspect birthmarks. However, this counterattack may not work in practice because (1) these injected system calls will most likely be filtered out in the noise filtering phase, and (2) system calls are very costly due to context switching between the kernel mode and the user mode. A large number of injected system calls will result in great slowdown of the plagiarism program, which thwart an plagiarist from using this

counterattack. Moreover, a bigger m can be always used to defeat such counter attacks. It is a tradeoff between performance and robustness.

Another possible counterattack is the system call reorder attack. Although this attack changes the order of system calls in traces, it leaves SCDGs unaltered. Two or more system calls can be reordered only when they are not bounded by dependences. Otherwise, reordering could break dependences, and thus cause behavior change or program errors. As such, this counterattack can evade our detection.

There may exist other counterattacks which overhaul SCDG birthmarks. For these unknown and unexpected attacks, SCDG birthmark is still robust to some extent, because it is tolerant to a certain percentage overhauling by using γ-isomorphism.

5.5.3 Limitations and Future Work

SCDG birthmark bears the following fundamental limitation. First, it will not apply if the program of interest does not involve any system calls or has very few system calls, for example, when there are only arithmetic operations in the program. Second, it is not applicable to the programs which do not have unique system call behaviors. For example, the only behavior of a sorting program is to read an unsorted file and print the sorted data. This behavior, which is common to other sorting programs or even irrelevant programs, is not unique. As such, our tool should be used with caution, especially for tiny common programs with few system calls. Third, as a detection system, it bears the same limitation of intrusion detection systems; that is, there exists a fundamental tradeoff between false positives and false negatives. The detection result of our tool depends on the parameters (m and γ) a user defines. To have higher confidence, one should use large parameters, thus it is likely to increase false negative. In contrast, reducing these parameters may increase false positive. Unfortunately, without many real-world plagiarism samples, we are unable to show some concrete results on such false rates although we have showed SCDG birthmarks exist for all the programs we studied. As such, rather than applying our tool to "prove" software plagiarisms, in practice one may use it to collect some initial evidences before taking further investigations, which often involve nontechnical actions.

We will study the impact of input on SCDG birthmarks as our future work. As a dynamic birthmark, SCDG birthmark requires the original program and the suspicious program to be fed with the same input. This requirement sometimes is difficult to meet. For example, a software plagiarist may illegally use a real time computer vision library as a part of their motion understanding software, whereas the original program uses the library for different purposes, say face recognition.

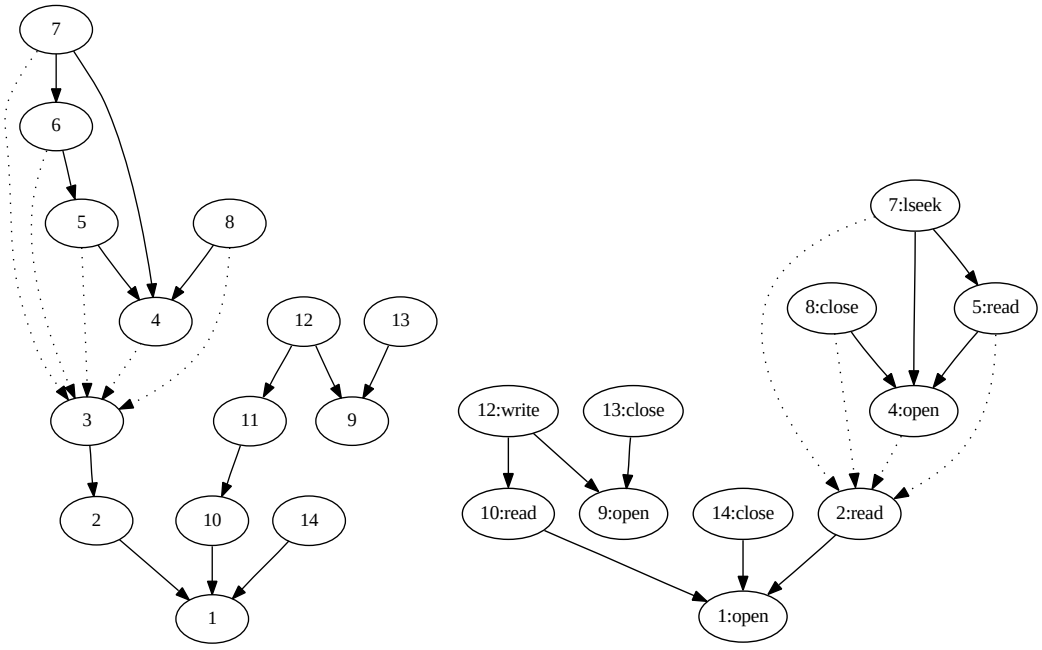
We will also examine how the birthmark of a component changes over software versions. A component of software might be upgraded frequently; some updates may change the implementation of the component significantly, which may include totally redesigned programming interfaces. These changes can invalidate the birthmarks computed before the updates. Will these changes help a plagiarist evade the detection by stealing an old version of the component? To answer this question, we will investigate the similarity of the birthmarks of different versions of the Gecko layout engine in the future.

5.6 Conclusion

We proposed the SCDG software birthmark. We evaluated it with a set of real world programs. Our experiment results showed that all the plagiarisms obfuscated by the two state-of-the-art tools were successfully discriminated. Unlike existing schemes that are evaluated with small or toy software, we evaluate our birthmarks with a set of large-scale software. The results showed that SCDG Birthmark is effective and practical in detection of components theft of large-scale programs.

```
1 :  fd1 = open(path1,"r",...);
2 :  read(fd1, buf, ...);
3 :  if (buf == "1") {
4 :      fd2 = open(path2, "r", ...);
5 :      n = read(fd2,buf,...);
6 :      offset = n + 10;
7 :      lseek(fd2,offset,...);
        ...
8 :      close(fd2);
     }
9 :  fd3 = open(path3,"w",...);
10:  read(fd1, buf, ...);
11:  strcpy(outbuf,buf);
12:  write(fd3,outbuf);
13:  close(fd3);
14:  close(fd1);
```

(a) statements



(b) DDG

(c) SCDG

Figure 5.1. An Example for DDG and SCDG

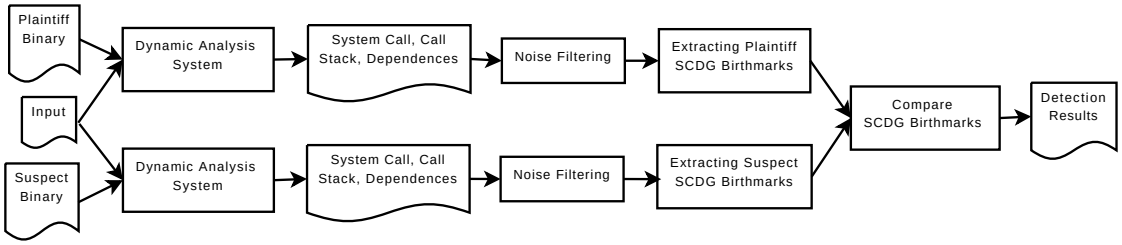


Figure 5.2. System Overview

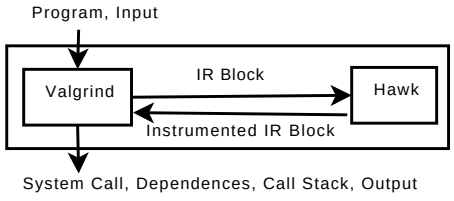


Figure 5.3. Dynamic Analysis System

```
0x4000B02: addl %edx,4(%eax)
—— IMark(0x4000B02, 3) ——
 *1:   t9 = GET:I32(0)              # get %eax
  2:   sh(t9) = sh(%eax)
 *3:   t8 = Add32(t9,0x4:I32)       # add address
  4:   sh(t8) = sh(t9)
 *5:   t2 = Ldle:I32(t8)            #load
  6:   sh(t2) = sh(memory(t8))
 *7:   t1 = GET:I32(8)              # get %edx
  8:   sh(t1) = sh(%edx)
 *9:   t0 = Add32(t2,t1)            # addl
 10:   sh(t0) = sh(t2) ∪ sh(t1)
*11:   STle(t8) = t0                # store
 12:   sh(memory(t8)) = sh(t0)

0x4000B05: movl 0x2E0(%ebx),%eax
—— IMark(0x4000B05, 6) ——
*13:   PUT(60) = 0x4000B05:I32      # put %eip
*14:   t11 = GET:I32(12)            # get %ebx
 15:   sh(t11) = sh(%ebx)
*16:   t10 = Add32(t11,0x2E0:I32)   # add
 17:   sh(t10) = sh(t11)
*18:   t12 = Ldle:I32(t10)          # load
 19:   sh(t12) = sh(memory(t10))
*20:   PUT(0) = t12                 # put %eax
 21:   sh(%eax) = sh(t12)
```

Figure 5.4. IR instrumentation Example. Statements with mark * are original IRs. Instrumentation IRs are pseudo code for brevity.

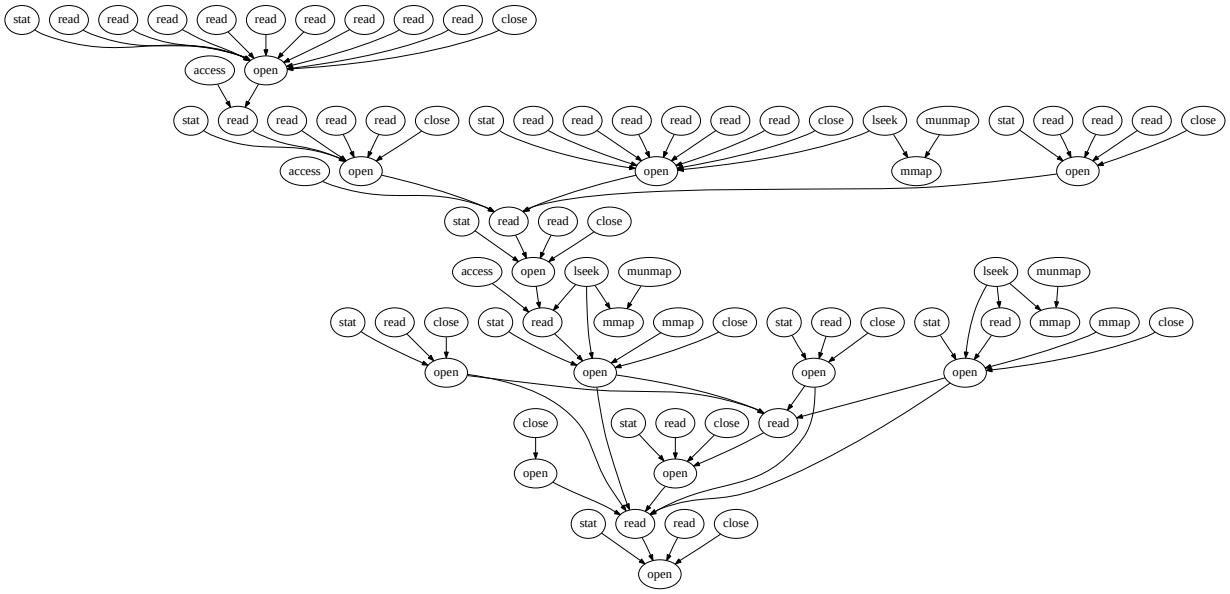


Figure 5.5. An Example Birthmark extracted From Aspell

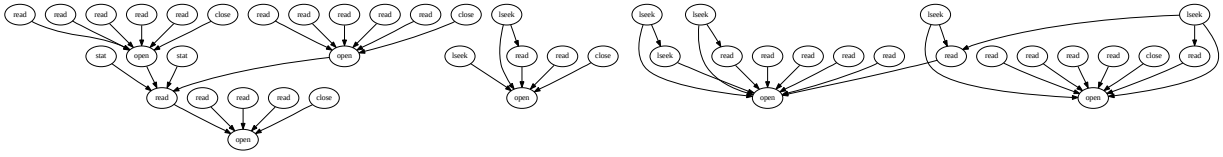


Figure 5.6. An Example Birthmark Extracted from Gecko

Chapter 6

Conclusions

In this dissertation, we design several approaches to protect software from attack and theft.

In Chapter 3, we propose a novel data-flow analysis technique called *code abstraction* to separate code from data. A detection system called SigFree based on this technique is designed and implemented. Because SigFree is signature free, it can block new and unknown buffer overflow attacks. We implemented and tested SigFree; our experimental study shows that the dependency-degree–based SigFree could block all types of code-injection attack packets tested in our experiments with very few false positives. Moreover, SigFree causes very small extra latency to normal client requests when some requests contain exploit code. SigFree is good for economical Internet wide deployment with very low deployment and maintenance cost, because it is a transparent deployment to the servers being protected,

In Chapter 4, a novel static taint and initialization analysis approach called STILL was presented, to address code obfuscation. Our static taint analysis technique enables STILL to collect strong evidence of self-modifying and/or indirect jump code obfuscation behaviors. As a result, while self-modifying code or indirect jumps may avoid the detection of other static analysis approaches, STILL can detect them with a high accuracy. Moreover, the static taint analysis technique turns out to be a comprehensive solution for detecting obfuscated code. For all the code obfuscation techniques we are aware of, STILL is immune to all but one. We implemented and evaluated STILL. Our experiment results show that STILL detected all the exploit codes generated by 10 well-known shellcode generation

engines and STILL achieves very few false positives in analyzing 378,158 HTTP requests/replies.

In Chapter 5, we proposed SCDG software birthmark to detect program component theft. We evaluated it with a set of real world programs. Our experiment results showed that all the plagiarisms obfuscated by the two state-of-the-art tools were successfully discriminated. Unlike existing schemes that are evaluated with small or toy software, we evaluate our birthmarks with a set of large-scale software. The results showed that SCDG Birthmark is effective and practical in detection of components theft of large-scale programs.

Bibliography

[1] http://news.zdnet.com/2100-3513_22-5629876.html.

[2] Buffer overrun in jpeg processing (gdi+) could allow code execution (833987). Available from World Wide Web: `http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx`.

[3] Citeseer: Scientific literature digital library. http://citeseer.ist.psu.edu.

[4] Computer emergency response team (cert). http://www.cert.org.

[5] Elsa: An elkhound-based c++ parser, http://www.cs.berkeley.edu/%7Esmcpeak/elkhound.

[6] The gecko engine, http://en.wikipedia.org/wiki/Gecko_layout_engine.

[7] Intel ia-32 architecture software developer's manual volume 1: Basic architecture. Intel Corporation. Available from World Wide Web: `http://developer.intel.com/design/pentium4/manuals/253665.htm`.

[8] Jempiscodes - a polymorphic shellcode generator. http://www.shellcode.com.ar/en/proyectos.html. Available from World Wide Web: `http://www.shellcode.com.ar/en/proyectos.html`.

[9] Khtml engine, http://en.wikipedia.org/wiki/KHTML.

[10] The metasploit project. http://www.metasploit.com.

[11] Microsoft security bulletin. http://www.microsoft.com/technet/security/current.aspx.

[12] Security advisory: Acrobat and adobe reader plug-in buffer overflow. http://www.adobe.com/support/techdocs/321644.html.

[13] Stunnel – universal ssl wrapper. http://www.stunnel.org.

[14] Symantec security response - backdoor.hesive. http://securityresponse.symantec.com/avcenter/venc/data/backdoor.hesive.html.

[15] the de facto standard for intrusion deetection/preventions. http://www.snort.org.

[16] Winamp3 buffer overflow. http://www.securityspace.com/smysecure/catid.html?id=11530.

[17] B. S. Baker. On finding duplication and near duplication in large software systems. In *Proc. of 2nd Working Conf. on Reverse Engineering*, 1995.

[18] H. G. Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29:29–9, 1994.

[19] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM conference on Computer and communications security*, October 2003.

[20] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Int. Conf. on Software Maintenance*, 1998.

[21] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard). Available from World Wide Web: `http://www.ietf.org/rfc/rfc1738.txt`. Updated by RFCs 1808, 2368, 2396, 3986.

[22] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security*, 2007.

[23] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[24] R. Chinchani and E. V. D. Berg. A fast static analysis approach to detect exploit code inside network flows. In *Recent Advances in Intrusion Detection Symposium*, 2005.

[25] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of ESEC/FSE*, 2008.

[26] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Int. Conf. on Softw. Maint.*, 1997.

[27] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2004.

[28] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999*, Jan. 1999.

[29] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, The Univeristy of Auckland, July 1997.

[30] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science,University of Auckland, 1997.

[31] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[32] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *ACM Symposium on Operating Systems Principles*, 2005.

[33] S. Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/0063.html.

[34] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. V. Underduk. Polymorphic shellcode engine using spectrum analysis. http://www.phrack.org/show.php?p=61&a=9.

[35] C. Economics. Virus attack costs on the rise. http://www.computereconomics.com/article.cfm?id=873.

[36] L. D. Fosdick and L. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8, September 1976.

[37] M. Hecht. *Flow analysis of computer programs*. Efsevier North-Holland,, 1977.

[38] C. Hoffman. *Group-theoretic Algorithms and Graph Isomorphism.* Springer Verlag, 1982.

[39] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *ACM Symp. on Theory of Computing*, 1974.

[40] J. E. Hopcroft and J. K. Wong. Performance evaluation of the vf graph matching algorithm. In *Processing of 10th Int. Conf. on Image Analysis and Processing*, 1999.

[41] J. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, 5(3), May 1979.

[42] C. Ionescu. Getpc code (was: Shellcode from ascii). http://www.securityfocus.com/archive/82/327348/2006-01-03/1, 2003.

[43] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7), 2002.

[44] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM conference on Computer and communications security*, October 2003.

[45] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Usenix Security Symposium*, 2004.

[46] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[47] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

[48] O. Kolesnikov and W. Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffic.

[49] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. In *Working Notes of 3rd Workshop on AI and Software Engineering*, 1995.

[50] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. of 8th Working Conf. on Reverse Engineering*, 2001.

[51] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection Symposium*, 2005.

[52] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, 2004.

[53] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote. Detecting and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11), 2005.

[54] Z. Li, M. Sanghi, Y. Chen, M. Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, May 2006.

[55] Z. Liang and R. Sekar. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *Annual Computer Security Applications Conference*, 2005.

[56] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *ACM Conference on Computer and Communications Security*, 2005.

[57] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conference on Computer and Communications Security*, October 2003.

[58] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006.

[59] S. Macaulay. Admmutate: Polymorphic shellcode engine. http://www.ktwo.ca/security.html.

[60] M. Madou, L. Van Put, and K. De Bosschere. Loco: An interactive code (de)obfuscation tool. In *Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, pages 140–144, 2006.

[61] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd ACSAC*, 2007.

[62] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. In *ISC*, pages 404–415, 2004.

[63] G. Myles and C. S. Collberg. K-gram based software birthmarks. In *SAC*, 2005.

[64] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*.

[65] J. Newsome, B. Karp, and D. Song. Polygraph: Automatic signature generation for polymorphic worms. In *IEEE Security and Privacy Symposium*, 2005.

[66] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium*, 2005.

[67] Noir. Getpc code (was: Shellcode from ascii). http://www.securityfocus.com/archive/82/327100/2006-01-03/1, 2003.

[68] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4), 2004.

[69] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In *GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2006.

[70] J. Poskanzer. http_load - multiprocessing http test client. http://www.acme.com/software/http_load.

[71] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *Universal Computer Science*, 2000. Available from World Wide Web: `citeseer.ist.psu.edu/article/prechelt01finding.html`.

[72] rix. Writing ia32 alphanumeric shellcodes. http://www.phrack.org/show.php?p=57&a=15, 2001.

[73] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[74] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for java. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007.

[75] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. In *IEEE Working Conference on Reverse Engineering*, 2002.

[76] S. Singh, C. Estan, G. Varghese, and S. Savage. The earlybird system for real-time detection of unknown worms. Technical report, University of California at San Diego, 2003.

[77] sk. History and advances in windows shellcode. Phrack, vol. 11, no. 62, 2004.

[78] skape. Understanding windows shellcode. http://www.nologin.org, 2003.

[79] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[80] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of java programs. In *Proc. IASTED International Conference on Software Engineering*, 2004.

[81] H. Tamada, K. Okamoto, M. Nakamura, and A. Monden. Dynamic software birthmarks to detect the theft of windows applications. In *International Symposium on Future Software Technology 2004 (ISFST 2004)*, 2004.

[82] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto. Design and evaluation of dynamic software birthmarks based on api calls. Technical report, Nara Institute of Science and Technology, 2007.

[83] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances in Intrusion Detection Symposium*, 2002.

[84] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[85] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM Conference on Computer and Communications Security*, 2005.

[86] K. Wang and S. J. Stolfo. Anomalous payload-based network instrusion detection. In *Recent Advances in Intrusion Detection Symposium*, 2004.

[87] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *15th Usenix Security Symposium*, July 2006.

[88] X. F. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: Black-box exploit detection and signature generation. In *ACM Conference on Computer and Communications Security*, 2006.

[89] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *ACM Conference on Computer and Communications Security*, 2005.

[90] O. Yuschuk. Ollydbg disassembler, 2006. Available from World Wide Web: `http://www.ollydbg.de/`.

[91] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Iyer. Analyzing network traffic to detect self-decrypting exploit code. In *AsiaCCS*, 2007.

Vita

Xinran Wang

Xinran received his MS and BE degrees in electrical engineering from Tsinghua University , Beijing, China, in 2001 and 1998, respectively. He worked as a software engineer for IBM's China Software Development Lab from 2003 to 2004 and Asiainfo Inc. from 2001 to 2003. He enrolled in the Ph.D. program in Computer Science and Engineering at The Pennsylvania State University in January 2005.